JYVÄSKYLÄ STUDIES IN COMPUTING

7

Minna Koskinen

Process Metamodelling

Conceptual Foundations and Application

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2000

JYVÄSKYLÄ STUDIES IN COMPUTING 7

Minna Koskinen

Process Metamodelling

Conceptual Foundations and Application

Esitetään Jyväskylän yliopiston informaatioteknologian tiedekunnan suostumuksella
julkisesti tarkastettavaksi yliopiston Agora rakennuksessa (Auditorio 2)
marraskuun 25. päivänä 2000 kello 12.

Academic dissertation to be publicly discussed, by permission of
the Faculty of Information Technology of the University of Jyväskylä,
in Agora (Auditorium 2), on November 25, 2000 at 12 o'clock noon.

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2000

Process Metamodelling

Conceptual Foundations and Application

JYVÄSKYLÄ STUDIES IN COMPUTING 7

Minna Koskinen

Process Metamodelling

Conceptual Foundations and Application

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2000

Editors
Seppo Puuronen
Department of Computer Science and Information Systems, University of Jyväskylä
Pekka Olsbo and Marja-Leena Tynkkynen
Publishing Unit, University Library of Jyväskylä

URN:ISBN:978-951-39-9046-6
ISBN 978-951-39-9046-6 (PDF)
ISSN 1456-5390

Jyväskylän yliopisto, 2022

ISBN 951-39-0823-2
ISSN 1456-5390

Copyright © 2000, by University of Jyväskylä

Jyväskylä University Printing House, Jyväskylä and
ER-Paino Ky, Lievestuore 2000

ABSTRACT

Minna Koskinen
Process Metamodelling: Conceptual Foundations and Application
Jyväskylä: University of Jyväskylä, 2000, 213 p.
(Jyväskylä Studies in Computing,
ISSN 1456-5390; 7)
ISBN 951-39-0823-2
Finnish summary
Diss.

This study deals with customisation of process modelling languages in method support technology. Technology plays an important role in process improvement since its capabilities limit the choices available for an organisation. The aim is to strive for technology that enables purposeful change, while avoiding technology that forces change to no purpose. The objective of this study is to develop a theory and mechanisms for support technology that enables language change. Process metamodelling is chosen as a means by which process modelling languages can be specified and implemented in a process support environment. The study forms part of a larger research effort on customisable method support environments.

The thesis studies the conceptual basis of process metamodelling and its application in metaCASE technology. The specific objectives are 1) to develop a system architecture for language specification and a generic process engine, 2) to investigate alternatives and principles for language specification, along with the use of these in process enactment, 3) to design and implement the constructs needed for language customisation in a generic modelling system, and 4) to design and implement the mechanism needed to enact process models in a generic process enactment system.

The research methodology takes a constructive approach. It proceeds through an incremental and iterative cycle of observation, theory building, system development, and experimentation. Prototyping forces the theory builder to experiment with the consequences of the theoretical assumptions present in experimental system designs. Each iteration increases the formality of the design, gradually improving and validating the theory. The research is finally synthesised in a set of criteria for assessing customisable method support environments.

Keywords: metaCASE, process support, method engineering, process engineering, process modelling languages, PML engineering

ACM Computing Review Categories

D.2.1. Software Engineering: Requirements/Specifications:
Languages, Methodologies, Tools
D.2.2 Software Engineering: Design Tools and Techniques:
Computer-aided software engineering (CASE)
D.2.10 Software Engineering: Management:
Software process models
D.2.11 Software Engineering: Software Architectures:
Data abstraction, Languages

Author's address

Minna Koskinen
Dept. of Computer Science and Information Systems
University of Jyväskylä
P.O.Box 35, SF-40350 Jyväskylä
Finland
e-mail: mkoskinen@acm.org
fax: +358 14 260 30311

Supervisors

Pentti Marttiin
Nokia Research Center, Helsinki
Finland

Kalle Lyytinen
Department of Computer Science and Information Systems
University of Jyväskylä
Finland

Reviewers

Klaus Pohl
Fachschaft Mathematik und Informatik
Universität GH Essen
Germany

Ilkka Tervonen
Department of Information Processing
University of Oulu
Finland

Opponent

Carlo Ghezzi
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Italy

ACKNOWLEDGEMENTS

I am indebted to many people and organisations for accomplishing this thesis. The thesis work was carried out at the Department of Computer Science and Information Systems, at University of Jyväskylä. I want to thank the many people that have helped me, in one way or another, to reach my goal. The funding for this study was provided by COMAS Graduate School, University of Jyväskylä, and Academy of Finland.

Prof. Klaus Pohl from Universität GH Essen in Germany and prof. Ilkka Tervonen from University of Oulu in Finland have acted as external reviewers of the dissertation. Their constructive comments and suggestions have helped me to improve the work in major ways.

I highly appreciate my head supervisor, Pentti Marttiin, for continued support and guidance. During the last six years, he has become a person whom I have learnt to trust and respect in many ways. He has allowed me great independence and responsibility, but he has also shared the research and co-authored four papers in the thesis. None of my problems has been too small or too much 'out of order' for him to discuss and to try to sort it out. As employed at Nokia Research Center, he has also given me a valuable empirical connection to the industry. It is my wish, and trust, that the co-operation will continue far into the future.

Prof. Lyytinen has kindly acted as my supervisor at the Department of Computer Science and Information Systems. I thank him for the possibility to work in MetaPHOR research group, and the many useful comments on my work. As one of the leading researchers in the field, with contacts to other leading researchers and research groups, he has been of great help merely by being there.

The MetaPHOR group has provided most favourable conditions for the study. I want to thank Steven Kelly, Matti Rossi and Juha-Pekka Tolvanen, who have given me valuable advice on conducting my work. I also recall the many interesting discussions with Janne Kaipala, Risto Pohjonen, Jouni Huotari and Zheying Zhang. Other members of the research group have included (in the order of appearance) Veli-Pekka Tahvanainen, Hui Liu, Juha Pirhonen, Harri Oinas-Kukkonen, Janne Luoma, Marko Somppi, Kalle Korhonen, and Matti Äijänen. Each of them has – undoubtedly – contributed to a stimulating and supportive atmosphere.

Furthermore, I want to thank Simo Rossi, Tero Sillander, and Mikko Kumpulainen at Nokia Mobile Phones/PMR in Jyväskylä. Observing their empirical research on PML engineering, process modelling and process support has given me valuable information and insight.

Finally, I want to thank my parents for encouraging me in my studies. It strikes me that the most useful skill I have needed in my work was taught by my father when I was little. When I asked him questions, the way little children do, he never gave me an easy answer. Instead, he returned the question and asked 'What do *you* think?'. Then, he paused his work and patiently waited

until I found a proper answer, guiding me with further questions when necessary. Thereby, I learnt how to study things carefully and how to use creative thought efficiently. Above all else, I learnt to share my father's deep passion and interest in 'studying all things', and to face intellectual challenges confident in my abilities to deal with them. In the doctoral study, I confronted the most interesting and the most puzzling challenge so far, and perhaps that is the reason why I haven't – quite unlike my habits – moved on a long time ago.

I also thank my sister Heli, especially for the well focused summer in Helsinki when I was finalising my licentiate thesis. Furthermore, I thank Merja for her unconditional support and friendship during these years.

Jyväskylä
October 2000

CONTENTS

Caelum, non animum mutant,
qui trans mare currunt.

– Horace

(Those who cross a sea change
the sky, not themselves.)

1 INTRODUCTION

1.1 Background and Motivation

Today, almost any research effort concerning systems development seems to be motivated by a desire for improving it. There is nothing peculiar in this, granted that the Puzzle of Systems Development – as the philosopher of science Thomas Kuhn would call it – has resisted all practical and academic attacks for the last four decades and, unfortunately, seems set to resist them well into the future. Research has brought forth countless innovations and improvements, but evidently not at the pace the requirements of systems development have evolved. Annoyingly enough, the innovations and improvements themselves seem to constitute a key motivator for new requirements.

It is characteristic of the last decades that they have trumpeted technical rationality as *the* management ideal of systems development, while confronted with overwhelming socio-cultural problems in practice. A major challenge that organisations face today is to create and maintain a balance between the instrumental-economic requirements of systems production and the socio-cultural requirements of human motivation. This is reflected in an increased interest in establishing connections to such fields as sociology and psychology.

The interest in improving systems development is, of course, an interest in quality. Through quality, a software organisation attempts to improve the satisfaction of its customers and thereby to maintain its competitiveness or simply to survive in the market. Yet, quality is a complex, multi-dimensional notion.

An interest in quality usually emphasises some specific motivation to quality. Firstly, an interest in quality is instrumental when it deals with the productive capabilities of an organisation. Thereby, systems development is viewed as an instrument for producing systems. Improvements in systems development aim to correct flaws in this instrument and make it more efficient and economical. Secondly, an interest in quality as a social concern deals with

the motivational capabilities of an organisation. Social quality can be seen as meaning that people in a software organisation are motivated and interested in their work – that they find their work socially rewarding. Thirdly, an interest in quality may also deal with quality improvement. It is manifested in the ability to adjust instrumental and social quality to suit the organisational context. Of primary importance in improvement therefore is that a software organisation can strike a balance between these different interests in quality.

An interest in quality often focuses on a specific sphere. The sphere of interest shows what aspects the motivation emphasises. Firstly, an interest in quality may concern technical issues. In systems development, technical consideration is usually given to software products and processes. Secondly, an interest in linguistic quality concerns the quality of communication. It may address the means and forms of interaction as well as the quality of languages themselves. Thirdly, an interest in quality is organisational when it concerns the interplay of organisational agents. Variation in the motivation and sphere of interest is reflected in which qualities are generally approved as indicators of quality. Examples of potential quality indicators are shown in table 1. The history of computer science and information systems research has demonstrated a slow but irrevocable transformation from a narrow, instrumental and technical concern towards a pluralist, more balanced view of quality.

TABLE 1 Some indicators of quality.

	Instrumental	Social	Improvement
Technical - product	reliability accuracy efficiency	usability feasibility satisfaction	changeability adaptability
Technical - process	predictability controllability measurability effectiveness	supportiveness convenience satisfaction ethicality	flexibility adaptability
Linguistic	formality expressiveness efficiency	comprehensibility equity self-expressiveness	reflection self-reflection conciliation
Organisational	control structure clarity	autonomy responsibility equality justice	learning enforcement empowerment emancipation

The goal of computer support in systems development is to improve quality by making methodical development more feasible. Attempts on this goal have followed two relevant research traditions. The method tradition has introduced systems development methods and CASE tools. Thereby, it has demonstrated a technical interest in system products. In contrast, the process tradition has focused on process improvement through process modelling and automate support tools. Its emphasis is thus on the technical part of the development

process. The early studies in both traditions were based on a narrow instrumental motivation. When the results were investigated empirically, researchers found persistent social opposition to their attempt to impose instrumental ideals. Finally this opposition led to research that recognised requirements and preferences that go beyond a purely instrumental motivation. This further motivated the emergence of research on customisation of method support technologies, and on customisable architectures.

The study presented in this thesis can be located at the intersection of the two traditions. It has grown up in the method tradition and reaches out to the territory of the process tradition. What it shares with both traditions is an interest in quality improvement. Where it moves into new territory is in its interest in linguistic quality. The study searches for a means to balance the instrumental motivation that almost invariably overpowers social concerns on the role of language in method support technologies.

The basis of this thesis is the recognition that – whatever else it may be and do – a computerised tool always implements a particular mode of thought. Technology, as it is introduced in an organisation, tends to change the way people comprehend their work. There are executives and managers that are concerned about this. They argue that current detail-intensive technologies have shattered employees' earlier holistic view of work that accounted in part for the success of the organisation. Technology providers have not recognised how strong an influence technology has on users' ways and modes of thinking.

The thesis focuses on process approaches implemented by process technologies. Any process approach imposes a specific model for process thinking. Process thinking articulates itself in the process modelling language and in the way that process support is implemented. An effort aimed at improving software development processes needs, to be successful, to recognise the cultural context and to make explicit the software practices as they are actually understood and applied by software developers (Sharp et al., 1999). A process approach should support the way in which people naturally conceptualise systems development and themselves as part of a systems development project. However, this social motivation should not be taken as implying that process approaches should not be designed, tailored and improved carefully and systematically. On the contrary, the clarity engendered by such an effort usually contributes positively to work motivation.

The positivistic ideals that have dominated Western thinking over the late century have appreciated and promoted a narrow instrumental interest, especially in technological research. As a consequence, support technologies tend to be implemented in conformity to some idealistic practice. Since there is no perceivable reason to change something that is ideal, no mechanisms for the adaptation and evolution are normally provided. Those who advocate this line of thought propose – explicitly or implicitly – that there exists one uniform and ideal way of thinking for different systems development efforts and organisations. As a result, the methods and processes supported constitute ideals that are difficult to obtain or faithfully follow in practice. This kind of thinking is strongly opposed in this thesis.

The study purports to increase the quality of customisable method support environments by increasing their capabilities for language change. The main contribution in this thesis is to introduce an approach to support language change in process support technologies. This approach is called process metamodelling. Process metamodelling is a means for the specification and profound adaptation of process approaches into a customisable process modelling and enactment system.

1.2 Research Background

The MetaPHOR group is a research group at the Department of Computer Science and Information Systems at the University of Jyväskylä in Finland (Lyytinen et al., 1994). The main goal of the group is to develop architectures, models and technical solutions for user-tailorable metaCASE environments, and principles for their effective use through method engineering. Since it was formed in 1989 it has conducted several projects in the field of method engineering. It has also developed two metaCASE environments, MetaEdit and MetaEdit+ (Kelly et al., 1996), both later commercialised.

Research on method customisation through process engineering began in 1994. The research on process engineering currently takes place in two locations. Firstly, research at the University of Jyväskylä includes theoretical and constructive studies on process engineering for requirements engineering and systems design in metaCASE. The emphasis of this research is on developing theories and architectures for constructing a generic process modelling and enactment system (Marttiin, 1998a; Koskinen, 1999). Secondly, research at Nokia Networks/PMR comprises empirical and constructive studies on process engineering for software design and implementation. The studies investigate various aspects of contextual adaptation and evolution for process modelling and process support in a software engineering project (Rossi & Sillander, 1998a). One of the main interests of this research is PML (process modelling language) engineering (Rossi & Sillander, 1998b).

The study presented in this thesis is carried out at the former location. The work at this location is conducted under the generic title "Process Engineering in metaCASE". The research has concentrated on the following four topics.

- *Architectural study* concerns integration of metaCASE and process support architectures to provide a more comprehensive architecture for customisable method support environments.
- *Process metamodelling study* investigates the specification and evolution of process modelling languages in a generic process modelling and enactment system.
- *Process modelling study* addresses the specification and evolution of process models in a generic process modelling system.
- *Process enactment study* investigates the design of generic enactment mechanisms to be implemented in a generic process enactment system.

The goal of the research position is to develop a generic process modelling and enactment architecture for user-tailorable process modelling and human-oriented process enactment. The objectives of the system are to support understanding, provide guidance for users, and co-ordinate modelling tasks.

The main difficulty faced in our earlier studies (Marttiin, 1994b) is the customisation of process modelling languages: how to increase the tailorability of process modelling languages in order to supply different projects with suitable process support. Such a capability is relevant when a process support tool is customised for many projects, or when a process approach will evolve within one project. Evolution of process modelling languages within a project is not widely studied and the available evidence of PML engineering does not consider tool support. However, a process support tool might be customised for projects in several organisations or for several projects within one organisation. In the former case, a PML engineer is an outside consultant who tailors process modelling languages for different companies, thereby lowering the threshold of adopting advanced technology. In the latter case, process modelling languages are customised to suit different project contingencies within an organisation.

1.3 Research Objectives and Questions

The main objective in this thesis is to develop a theory for applying metamodelling in the specification of process modelling languages. Along with this objective it also studies the mechanisms for using such specifications in a generic process modelling and enactment system. The aim of the system is to support rapid prototyping of process modelling languages. Changes should also be allowed during process enactment. The study develops a conceptual model and a related tool set for process metamodelling, along with enactment mechanisms that can cope with arbitrary process modelling languages.

The specific objectives of this thesis are: 1) to develop a system architecture for PML specification and a generic process engine; 2) to investigate alternatives and principles for PML specification and for the use of language specifications in process enactment; 3) to design and implement the language constructs needed for PML customisation in a generic modelling system; and 4) to design and implement the enactment mechanism needed to enact process models in a generic process enactment system. These objectives yield the research questions listed in table 2.

TABLE 2 The research questions addressed in this thesis.

Question 1	Architectural principles • What kinds of architectural principle are there for PML specification? • What kinds of architectural principle are there for a generic enactment mechanism?

(continues)

TABLE 2 (continues)

Question 2	Alternatives and principles • What alternatives and principles are there for PML specification? • What alternatives and principles are there for the use of PML specifications in process enactment?
Question 3	Language constructs • What kinds of language construct are needed in PML customisation? • How are these language constructs implemented in a generic modelling system?
Question 4	Enactment mechanisms • What kinds of enactment mechanism are needed to enact process models in a generic process enactment system? • How can the enactment mechanisms be implemented in a generic enactment system?

1.4 Research Methodology and Research Process

The significance of the contextual nature of language and other cultural issues is currently not acknowledged when building automated support for systems development. This has a two-fold implication on this study. On one hand, we should study how such factors affect the design of a support system before we design one. On the other hand, we do not have the necessary platform to study those effects unless we implement one. Therefore, we have chosen to use prototyping as part of the research. Through an incremental research approach, we attempt to develop a consistent theory for process metamodelling and an architecture for a generic process modelling and enactment system.

1.4.1 Research methodology

The research methodology consists of a constructive approach, in which research proceeds through an incremental and iterative cycle of observation, theory building, system development, and experimentation (Nunamaker et al., 1991). Firstly, observation in this study is based on a case study conducted at Nokia Mobile Phones/PMR. This opportunity was offered us while we were conducting the later cycles of the study. Hence observation has mainly taken a guiding role in the study. Secondly, theory building is based on prior metaCASE research and process studies in the MetaPHOR group, extended with extensive literature reviews. During the later cycles, observation, the prototyping experiment, and several design experiments have also contributed to theory building. Thirdly, system development in the form of prototyping has played an important role. Prototype development serves both as feedback and proof-of-concept, and provides a baseline for further research. Fourthly, experimentation has been carried out both with the prototype and a design environment.

Experimentation with the prototype has been three-fold. Firstly, we performed experiments to ensure that customised process metamodels can be used to configure the generic process editor and to model processes. Secondly, we performed experiments to ensure that customised process metamodels can be used to configure the generic process engine and to enact process models. Thirdly, we performed experiments with PML design to test the metamodelling capability. Thereafter, we performed some experiments to improve the metamodelling capability. We implemented the GOPRR-p model as a modelling technique in MetaEdit+ and used this technique to design the conceptual framework of several process modelling languages. This allowed us to improve the GOPRR-p model and test the changes immediately. This in turn made possible a rapid prototyping approach for design and validation of the improvements made in the GOPRR-p model.

1.4.2 Research process

The general outline of the resulted research process is illustrated in figure 1. The research started with an initial theory building phase that involved a literature review of process modelling languages. It was followed by prototype development that was conducted in two phases of design, implementation, and testing (Chapter 5 and 6). Each testing stage contributed to further theory building, and some initial design experiments were carried out. The development was iterated until the prototype was considered adequate for more comprehensive design experiments.

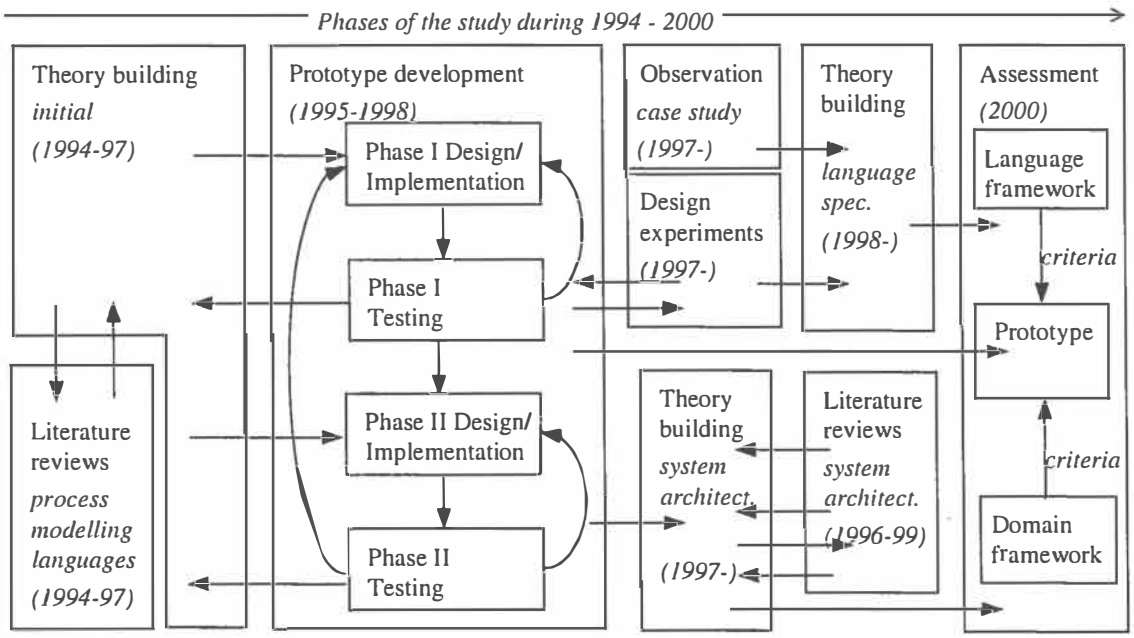

FIGURE 1 The research process in this study.

Simultaneously with these experiments, we had an opportunity to follow a related case study conducted in a software development organisation. The design experiments and observations contributed to further theory building (Chapter 3). The theory building phase focused on process metamodelling and resulted in a generic language model (Chapter 4).

Meanwhile, the experience gained from developing the prototype led to theory building concerning relevant system architectures. Literature reviews were conducted that contributed to theory building and the development of a general architecture for a customisable design environment (Chapter 2).

The language model and the general architecture formed the basis for a domain framework. This framework contains a set of assessment criteria for customisable method support systems. The prototype was assessed against these criteria to reveal areas for further development (Chapter 7).

1.4.3 Validation in the Research Approach

The research approach uses prototyping as part of the research method. In such an approach, the question of validity necessarily becomes a target of special inspection. There are two common approaches for ensuring the validity of constructive research. Firstly, prototyping may be used to demonstrate the feasibility of a proposed implementation approach for a theory validated earlier. An important part of validation is formalisation. Secondly, prototyping may be used as a means for theory validation. The study involves the use of the prototype in laboratory or field experiments to test its usability. Claims for the validity of the theory are based on the results of the experiment. In this study, we use prototyping in neither of these ways. We do not have a validated theory as a basis for the prototype, do not formalise it, and do not use it in laboratory or field experiments to test its usability. The subject of the study is such an abstract one that comprehensive theory building and its valid operationalisation for meaningful laboratory and field experiments in a prototypical tool takes enormous amount of time. Thus, we have to place more emphasis on the method of theory building and operationalisation and thereby attain a certain degree of validation in the method.

The research approach could be called self-validating constructive research: although prototyping plays a central role in the process, there is no claim for its validity in any phase of the research. Instead, prototyping is a method that forces the theory builder informally yet in a very detailed manner to experiment with the consequences of certain theoretical assumptions present in experimental system designs. Such a research process can not be a straight-forward process that begins from theory building and ends with prototype implementation.

Prototyping requires a prior, extensive theory building phase that uses different qualitative methods. In this study, a literature review was used in which about 200 relevant research articles or other publications were examined.

Prototyping was divided into two iterative phases, each of which consisted of system design, implementation, and testing. Each phase tested the

experimental system design and the results of the tests provided feedback to theory building. An important part of theory building was the analytical and systematic examination of the proposed experimental system designs. An experimental system design must meet certain generic design principles and criteria, such as conceptual clarity, comprehensiveness, and no conceptual redundancy. A design decision requires conceptual justification for all conceptual discriminations and integration. Prototyping is divided into phases to limit the complexity and scope of the experimental designs and thus to make conceptual examination more efficient and less error-prone. Towards the end of prototype development, we introduced experimental language designs in the method. Language designs iterated through the same conceptual analysis as system designs.

It is necessary that theory building and prototyping proceed cyclically. The validity of the study increases gradually as the study passes through several cycles of conceptual analysis and system design experimentation. The cycles are iterated as long as conceptual weaknesses are detected. Although the cyclical process improves the validity of the theory that is built alongside prototyping, the validation process should not end with it. Despite systematic conceptual analysis, a prototype easily makes its developer blind to its faults.

Therefore, one must introduce a means to experiment with the design apart from the prototype. As discussed above, we implemented the GOPRR-p model as a modelling technique in MetaEdit+ and used this technique to design the conceptual frameworks of several process modelling languages. These experimental designs too passed through iterative conceptual analysis. As a customisable design system, MetaEdit+ allowed us to improve the GOPRR-p model and test the changes immediately. We found this rapid prototyping approach valuable in designing and validating the improvements made in the GOPRR-p model.

This research approach presents a self-validating process in which each design and implementation iteration increases the formality of the design. Although the approach does not use formalisation as a part of the method, it systematically forces the researcher's thinking towards increasing formality. The iterations force the researcher to think and rethink the theory and designs in detail. In the process, the researcher may become so familiar with the details of the design that he or she can "debug" the design just by thinking about it.

After finishing prototype development, we initiated another process of validation. In contrast with the first, analytical, method, the second method is based on creating syntheses. The conceptual analysis in this phase emphasises conceptual discrimination, rearrangement, and integration. The phase requires extensive use of qualitative methods. We used literature reviews and a case study. The advantage of qualitative methods is that they provide new insights and necessitate the use of interpretation: prerequisites for conceptual synthesis. Qualitative methods make synthesis easier. An important point is that synthesis should be formed on the base of prior analysis, since this makes synthesis more robust and thus increases its validity.

The third phase of validation consists of creating a set of assessment criteria for assessing constructions of which the prototype is an example. The

criteria are based on an improved understanding of the target of research and the research area. This phase yields the most important scientific contribution. The assessment criteria are explicit and can be examined by the research community. They can also be reused in other similar assessments. Thus they enable future generalisation from the results of the study.

1.5 Introduction to the Paper Chapters

This thesis includes an introduction and six research papers, each of which constitutes one chapter. The papers are grouped into four parts:

I	Background	(Chapter 2);
II	Theory	(Chapter 3 and 4);
III	The CPME Prototype	(Chapter 5 and 6); and
IV	Assessment	(Chapter 7).

The research presented in this thesis has been carried out in a research group where several researchers study largely overlapping issues. The greatest benefits of such a research environment are that it supports collective evolution of ideas and accumulation of findings and knowledge of the research field. It also aids in the research work in many practical ways. While it is perhaps an ideal setting for conducting research, it also entails problems for compiling and defending a thesis. Firstly, it is not possible to isolate one's work for presentation, and secondly it is difficult to demonstrate one's contribution to the study.

Nevertheless, I have attempted to reduce this problem by collecting a set of papers in which my contribution is most substantial. First, the core contribution in each paper concerns my personal research work. Secondly, in all the papers I am either the only author or the lead author responsible for compiling and editing the paper. In the three joint papers (Chapters 2, 5 and 7) I have been responsible for the greatest part of both research and writing.

The following summaries briefly outline the core contribution of each paper, and illustrate how they contribute to the research questions in Section 1.3. They also identify my personal contribution to the joint papers and acknowledge co-authors and other main contributors.

Part I: Background

Chapter 2 presents a paper entitled "Comparing Two Traditions: Towards an Integrated View of Method Engineering and Process Engineering." The paper presents a detailed review and comparison of method engineering and process engineering research. It provides definitions for the core terms used in this thesis. It also shows how the relevant research areas are related. The paper contributes to Question 1 on the architectural principles.

The paper is a joint article with Pentti Marttiin. My contribution to the paper is the analysis of existing research and the compilation of the proposed architecture. The paper will be submitted to ACM Transactions on Software Engineering and Methodology.

Part II: Theory

Chapter 3 presents a paper entitled "Towards Customisation of Process Modelling Languages in Computer Aided Process Engineering." The paper illustrates different forms of linguistic adaptation in process support environments and develops the concepts of language adaptation and PML customisation. It also discusses PML engineering and its relation to process engineering. The paper contributes to Question 2 on the alternatives and principles for language specification and the use of such specifications in process enactment.

The paper has been submitted to the 23rd International Conference on Software Engineering (to be held in Canada, June 2001).

Chapter 4 presents a paper entitled "Conceptual Foundations of Process Metamodelling." The paper develops a theory of process metamodelling and illustrates a design of a comprehensive model of process modelling languages. It also discusses a future extension of the design to process modelling techniques with an operational model. The paper contributes to Question 2 on the principles and Question 3 on the language constructs needed in language specification.

This paper has been submitted to the ACM Transactions on Software Engineering and Methodology.

Part III: The CPME Prototype

Chapter 5 presents a paper entitled "Developing a Customisable Process Modelling Environment: Lessons Learnt and Future Prospects." The paper presents the architecture and components of CPME, and discusses its objectives in organisational support and evolution. The paper contributes to Question 1 on architectural principles.

The paper is a joint article with Pentti Marttiin. My contribution to the paper is the elaboration of CPME's role as organisational technology by pointing out some important issues in initial phase process improvement. The paper is published in the proceedings of the 6th European Workshop on Software Process Technology, EWSPT'98 (Koskinen & Marttiin, 1998).

Chapter 6 presents a paper entitled "Process Support in MetaCASE: Implementing the Conceptual Basis for Enactable Process Models in MetaEdit+." The paper presents the design and implementation of the GOPRR-p model and metamodelling tools in the CPME prototype. The paper contributes to Question 3 on the language constructs and their implementation.

This paper is a joint article with Pentti Marttiin. My contribution to the paper is the detailed design and implementation of GOPRR-p and the process metamodelling tools. The paper is published in the proceedings of the 8th Conference on Software Engineering Environments (Koskinen & Marttiin, 1997).

Part IV: Assessment

Chapter 7 presents a paper entitled "A Generic Process Modelling and Enactment System: Implementation and Assessment." The paper describes the CPME prototype in detail, and assesses CPME/MetaEdit+ against a set of criteria developed for customisable method support environments. The paper contributes to all the research questions, especially Question 1 on architectural principles, and Question 4 on enactment mechanisms and their implementation.

The paper is a joint article with Pentti Marttiin. My contribution to this paper is the design and implementation of the GOPRR-p model and the generic process engine. I have also developed the domain framework and the assessment criteria and used them to assess CPME/MetaEdit+ A shorter version of the paper will be submitted to the IEEE Transactions on Software Engineering.

A word of warning is also appropriate, since the papers have been written at different times. The ideas presented and the terminology used in the earlier papers have somewhat changed to reflect the improved conceptualisation and understanding of the subject. Unfortunately, copyright restrictions for published papers prevent the author from updating the ideas and terminology, and improving the clarity of writing. There are also differences in connotation and emphasis that sometimes act as a reason to use different terms.

1.6 Overview of the Work

In a thesis compiled of a collection of distinct papers, the presentation of the work tends to be scattered. Thus, it may be hard for the reader to build up the overall picture merely by reading the paper chapters. I therefore attempt to synthesise an overview of the work with references to distinct chapters.

Context. The subject of this thesis is located in the cross-section of method engineering and process engineering (see Chapter 2). A general framework for customisable design environments is presented (page 52). In terms of the framework, the area targeted in this thesis is *customisable CAPE and its reflection on CAPE and PCSE*. In Chapter 7, another framework is presented that gives a

more detailed view of customisable method support environments[1] (page 198). In terms of this framework, the area can be defined as *technique specification (for process modelling techniques) and its reflection on process modelling and enactment.* Chapter 3 is an introduction to this area with discussion on PML customisation and PML engineering.

Theory. The core theoretical work is discussed in Chapter 4. It identifies process modelling languages as parts of process modelling techniques, and process metamodelling as a means of their customisation. It develops the theory into a conceptual model of process metamodels, a "conceptual process meta-metamodel". The model distinguishes between conceptual, notational, and semantic information in a language specification, and operational information in a technique specification. These types of information are reflected in process modelling and process enactment.

Application. The application of the process metamodelling approach in the CPME prototype[2] is discussed in Chapters 5, 6 and 7. Chapter 7, in Section 2, gives the most comprehensive view of the prototype. The prototype implements a process meta-metamodel and process metamodelling tools (Chapter 6) for the specification of process modelling languages in process metamodels (see also page 185 in Chapter 7). The process metamodels are further used as PML specifications for a generic process engine. This process engine combines the functionality of a metaengine and an ordinary process engine. Conceptual and notational information is used in process modelling, and semantic information in process enactment.

1.7 Conclusion

The study aims to increase the quality of customisable method support environments by increasing their capabilities for language change. Specifically, it introduces a metamodelling approach concerned with linguistic change in process support technologies. Process metamodelling is a means for the specification and profound adaptation of process approaches in a customisable process modelling and enactment system.

The contributions of this thesis can be divided into two groups. First, a large part of the theory-related contribution is formed of several classifications, and philosophical and conceptual clarification of the studied phenomenon and its context. Such work forms an important part of any scientific effort, but especially when new or marginal issues are studied. In this thesis, I have extensively explored the context and the background of my specific subject. I have clarified the relation between method engineering and process

[1] The term "customisable design environment" in Chapter 2 is used as a general term that also covers "customisable method support environments". The use of the latter term reflects a narrower emphasis on customisability.

[2] The reader should, however, take into account that the prototype reflects an earlier stage in theory development than the one discussed in Chapter 4. As earlier publications, the contents of Chapters 5 and 6 are therefore not fully compatible with the theory presented in Chapter 4.

engineering by comparisons and definitions. This contributes to further integration of the traditions. The insights regarding the architecture of customisable method support environments, and the criteria for assessing them, are especially useful.

Second, a great emphasis in this thesis is laid on understanding and clarifying the nature of modelling languages and techniques, and different forms of metamodelling. This has been a necessity, since current studies do not give a proper foundation to apply in process metamodelling. Hence, I have been compelled to substantially extend previous studies in these areas. My special interest, nevertheless, has been to develop a means for PML customisation in a method support environment: "process metamodelling". I have extended the theoretical work on modelling languages and metamodelling to suit the needs of process modelling and enactment. As a result, a conceptual model of process metamodels has been developed, together with considerations on its application. The process metamodelling theory enhances metaCASE research with an approach to process support, and enhances PCSE research with an approach to PML customisation.

Third, this thesis addresses the architecture and design of a process metamodelling system and a generic process enactment system. Part of the contribution is the design and implementation of the GOPRR-p model, Process Metamodelling Tools, and a generic Process Engine in the CPME prototype. However, a more significant contribution is made by further theory building and critical assessment of the prototype. To this end, I have developed a domain framework with various criteria for the assessment of customisable method support environments. This provides a baseline for further research and development of such architectures.

There are several recommendations that can be made based on this study. Firstly, we recommend that researchers in method engineering and process engineering areas change their perspective from a technical one to a systemic one. There is a vast amount of research on information systems, the results of which could benefit these two areas. Furthermore, we recommend that researchers in each area become more concerned with research conducted in the other area. The present work has given some directions for possible contribution. Secondly, more research on the local adaptation and customisation of process modelling languages and techniques should be conducted. Especially, researchers should study the detailed architecture and design of customisable CAPE environments that would allow linguistic change as a natural part of process improvement. The domain framework and the assessment criteria constitute some guidelines for customisable system architectures. We also find that the development and comparison of process modelling languages should be made more systematic. In this regard, the theoretical considerations on process modelling languages presented in this thesis will be useful.

References

Kelly, S., Lyytinen, K. & Rossi, M. 1996. METAEDIT+ — A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In P. Constantopoulos, J. Mylopoulos & Y. Vassiliou (Eds.) Advanced Information Systems Engineering, LNCS 1080. Berlin: Springer-Verlag, 1-21.

Koskinen, M. 1999. A Metamodelling Approach to Process Concept Customisation and Enactability in MetaCASE. University of Jyväskylä. Computer Science and Information Systems Reports, Technical Reports TR-20. Jyväskylä. Licentiate thesis.

Koskinen, M. & Marttiin, P. 1997. Process Support in MetaCASE: Implementing the Conceptual Basis for Enactable Process Models in MetaEdit+. In J. Ebert & C. Lewerentz (Eds.) Software Engineering Environments. Los Alamitos: IEEE Computer Society Press, 110-123.

Koskinen, M. & Marttiin, P. 1998. Developing a Customisable Process Modelling Environment: Lessons Learnt and Future Prospects. In V. Gruhn (Ed.) Proceedings on the 6th European Workshop on Software Process Technology, EWSPT'98, LNCS 1487. Berlin: Springer-Verlag, 13-27.

Lyytinen, K., Kerola, P., Kaipala, J., Kelly, S., Lehto, J., Liu, H., Marttiin, P., Oinas-Kukkonen, H., Pirhonen, J., Rossi, M., Smolander, K., Tahvanainen, V.-P. & Tolvanen, J.-P. 1994. MetaPHOR: Metamodeling, Principles, Hypertext, Objects and Repositories. University of Jyväskylä. Computer Science and Information Systems Reports, Technical Report TR-7. Jyväskylä.

Marttiin, P. 1994. Towards Flexible Process Support with a CASE Shell. In G. Wijers, S. Brinkkemper, T. Wasserman (Eds.) Advanced Information Systems Engineering, LNCS 811. Berlin: Springer-Verlag, 14-27.

Marttiin, P. 1998. Customisable Process Modelling Support and Tools for Design Environment. University of Jyväskylä. Jyväskylä Studies in Computer Science, Economics and Statistics 43. Jyväskylä. PhD Thesis.

Nunamaker, J.F. jr., Chen, M. & Purdin, T.D.M. 1991. Systems development in Information Systems Research. Journal of Management Information Systems, 7, 3, 89-106.

Rossi, S. & Sillander, T. 1998a. A Software Process Modelling Quest for Fundamental Principles. In R. Walter & J. Baets (Eds.) Proceedings of the 6th European Conference on Information Systems (ECIS). Spain: Euro-Arab Management School, 557-570.

Rossi, S. & Sillander, T. 1998b. A Practical Approach to Software Process Modelling Language Engineering. In V. Gruhn (Ed.) Proceedings on the 6th European Workshop on Software Process Technology, EWSPT'98, LNCS 1487. Berlin: Springer-Verlag, 28-42.

Sharp, H., Woodman, M., Hovenden, F. & Robinson, H. 1999. The Role of 'Culture' in Successful Software Process Improvement. In: G. Chroust (Ed.) Proceedings of the 25th Euromicro Conference (EUROMICRO '99), Milan, Italy, September 8-10.

PART I: BACKGROUND

2 COMPARING TWO TRADITIONS: TOWARDS AN INTEGRATED VIEW OF METHOD ENGINEERING AND PROCESS ENGINEERING

Koskinen, M. & Marttiin, P. "Comparing Two Traditions: Towards an Integrated View of Method Engineering and Process Engineering".

This paper has been submitted for publication. Copyright may be transferred without further notice and the accepted version may be posted by the publisher.

Comparing Two Traditions: Towards an Integrated View of Method Engineering and Process Engineering

Minna Koskinen
University of Jyväskylä

Pentti Marttiin
Nokia Research Center

Abstract

The question of how to develop systems and software in a more disciplined way has exercised the minds of researchers for several decades. A number of exact methods and processes have been introduced. Studies on the benefits of such disciplined approaches recurrently present conflicting results, except the conclusion that no universal approach suits all situations. In consequence, two higher level engineering traditions have arisen but grown separately. Method engineering and process engineering are overlapping and complementary, yet there is little research on their relationships. We find that the traditions can help each other to reshape and better understand themselves. Consequently, this study aims at a more comprehensive and balanced view of methods and method research. Thereby it contributes to the further integration of the traditions. This article provides an integrated view of method and process engineering. We discuss the approaches and present them in a manner in which similarities and differences are easy to recognise. The study is aimed at researchers and tool providers in the new millennium. We expect method engineering and process engineering to become closer, thereby providing new flexible ways of working and new platforms for tools.

1 Introduction

Research on systems development methods was initiated to improve quality in systems development. The aim was to extract and codify successful practices and thereby to systematise the conduct of systems development. The earliest approaches to method development introduced common techniques for systems specification (Dijkstra, 1969; Yourdon and Constantine, 1979; Yourdon, 1989). Many computer tools were developed for the support of these methods (Waters, 1974; Teichroew and Hershey, 1977). Also, more comprehensive methods were introduced (Auramäki et al., 1992). Some research was also dedicated to systems development processes and the co-ordination of process actors (Royce, 1970; Baker, 1972). The early process models aimed at supporting communication between actors and contributed to a deeper understanding and learning of the process.

1.1 Two traditions

In the late 1980's, two traditions emerged that shortly took separate courses. Within the "method tradition", research on method support technology was conducted to develop an architectural framework that would integrate different techniques and tools within one environment. Techniques as notations and metadata became the core of method implementations.

Individual methods were found to be applicable only for certain purposes and more or less adapted to local practices (Pyburn, 1983; Wijers and van Dort, 1990; Aaen et al., 1992). This shifted the interest towards contingency and customisation approaches. The first method providers had offered standard solutions in text-books and methodically "fixed" tools. Therefore it was necessary to try to identify those development contingencies that would predict the suitability of available methods and that could be used in local method selection.

Method engineering and customisable tools emerged to provide further flexibility: to enable the design and construction of local methods (Bubenko, 1988; Heym and Österle, 1993). Later, the finding that method requirements change as users' understanding accumulates through methods' use, directed research towards incremental method engineering (Tolvanen, 1998).

Elsewhere, a significant milestone showed way for the "process tradition". This was Osterweil's (1987) proposition that the software development process could be automated. The 'process' became a means of method integration, with which to manage the use of individual techniques and tools. Thereby, it played a core role in method implementation.

Also process research confronted the need to locally adapt and evolve processes and tools. The text-book approach to processes was largely rejected after some attempts to develop standard processes (Royce, 1970). Contingency approaches to select among possible process alternatives did not gain much interest. Instead, process modelling was introduced as a means to specify local processes (Curtis et al., 1992), thereby leading research to process engineering approaches. The interest in process improvement models also increased (Paulk et al., 1993; Dorling, 1993; Haase et al., 1994). Process support technology was developed to enhance the efficiency of using process models. Towards the mid 1990's, studies on process evolution introduced new mechanisms for customising process technology (Madhavji, 1991; Bandinelli and Fuggetta, 1993; Kaiser and Ben-Shaul, 1993).

It can be said that the 1980's was the 'golden decade' of methods, while processes took over the next one (DeMarco, 1996). The "method jungle" of late 1980's (Avison and Fitzgerald, 1988) grew up an abundance of methods, some of which have merged or diversified, developed and survived better than others. The 'process jungle' is similarly a phenomenon of today. Neither tradition has yet established itself in the industry. However, an increasing interest in method customisation and technology is evident. Practitioners manifest interest also in process improvement, whereas process technology has so far shown little success (Conradi et al., 1998).

1.2 Towards the merge of traditions

One of the greatest shortcomings of the last decade has perhaps been the two traditions drifting apart. Contacts between the traditions have been weak or non-existing. This is shown by the fact that some major research questions of one tradition today are essentially the same as those of the other right after the traditions took separate courses. Yet this is hardly noticed.

The reason for that the traditions have not interacted properly during their history may be that the traditions have long focused on different phases of systems development. Method tradition has mostly concentrated on the early phases of systems development, such as systems analysis and design, with the need to manage complex and sensitive system requirements. In contrast, process tradition has concentrated on the later phases, such as software design, implementation, and testing, with the need to control and automate routine tasks (Curtis et al., 1992; Armenise et al., 1993; McChesney, 1995). Interest in the early processes has recently increased (Wijers, 1991; Harmsen et al., 1994a; Jarke et al., 1994; Rolland et al., 1995; Pohl, 1996). The focus is on user centred approaches: guidance and control mechanisms, learning support through process models and process traces, and process improvement through accumulated knowledge.

The traditions' interest in each other's findings seems mostly superficial, and the lack of interaction manifests itself in the research conducted to date. On one hand, some frameworks consider products only and focus on notations and metadata (Bergheim et al., 1989; ISO/IEC, 1990). Until recently, this has been characteristic to the method tradition. Some frameworks, on the other hand, consider processes only (McChesney, 1995; Lonchamp, 1993). Although they include the notion of product or alike, the properties of products are determined solely from the role of the products in a process (e.g., owner, size, creation-date). This is characteristic to the process tradition.

There are also frameworks, in which both the product and the process viewpoint are considered. These have mostly emerged within the method tradition as researchers have found a limited product viewpoint insufficient. However, these integrated approaches still have shortcomings. First, some frameworks have a weak notion of product. One potential shortcoming is that tools determine the metadata. Although the metadata model of a repository is customisable, tools are not. Instead, the metadata model is customised according to selected tools (Pohl and Jarke, 1992). When the operations provided by tools are customisable, the metadata is not (Pohl et al., 2000). Another potential shortcoming is the low internal integrity of metadata (Heym and Österle, 1993). Techniques are seen as manipulating loosely related conceptual and notational components. The metadata integrity is too low for building products methodically without continuous process support. Second, some frameworks have a weak notion of process (Harmsen et al., 1994a). The view of process is narrow and only limited forms of support can be provided.

Little research is conducted to understand the relationship between method engineering and process engineering. Frameworks are developed for one tradition and often explicitly oppose the other. It has remained unnoticed –

not only that these traditions could and should be integrated – but also that the two traditions can help each other to reshape and better understand themselves.

This study aims to get the traditions somewhat closer. We have used the following review method. First, we constructed a profile of both traditions. The profiles concentrate on issues that have interested respective researchers. Thereafter, we compared these profiles and matched similar issues in both traditions. The issues that remained in one tradition, showed potential gaps in the research carried out within the other. We considered how these 'missing' issues appear within the tradition and developed thereby a series of important insights. Through comparing different perspectives we were able to reshape both views and to reveal their close relationship. We regard this study as a necessary step to enable a more detailed inquiry into the integration of method engineering and process engineering.

In the following, we illustrate the two views with several definitions based on the comparisons. These definitions form a framework that serves as a basis for further research. A guiding notion throughout this study is two views of method promoted by the methodical traditions. (Section 2.2). We discuss method engineering and process engineering (Section 2.3), method modelling (Section 2.4), and technology for method use and customisation (Section 2.5). As a result of this study we outline a comprehensive design environment architecture and discuss five strategic integration points. Thereafter, we discuss its strategic integration points (Section 2.6). Finally, we summarise the main findings of this study (Section 2.7).

2 Two Views of Method

Systems development is an approach to produce information systems, in which a development group pursues to achieve some objectives using a systems development method. A method is a collection of guidelines, procedures, techniques and tools for developing information systems, based on a particular philosophy of systems development and of a particular system domain (Wynekoop and Russo, 1993). Methods serve as a means to better understand and produce the target system, to manage the overall development effort, and to aid communication and organisational learning. They are expected to lead to more acceptable and successful solutions, and to a better-managed development process (Tolvanen, 1998). The approaches to methods differ in regard to the focus of methods, how they are integrated, and the form of method rules.

To best characterise the method tradition is to say that it is product-centred. The tradition emphasises the structure and organisation of the products of systems development. Methods and techniques make explicit the required characteristics of a product. They use declarative rules to specify what kind of model is desirable or required, and they do not (usually) constrain the

actual steps taken to construct one. Method integration clarifies the structural dependencies that may combine models created with different techniques. The aim of method integration is to increase coherence and accessibility of model components while at the same time to reduce their overlap and redundancy.

Definition 1 A *product-centred method* is a method that concentrates on the structure and organisation of products and uses product-centred metrics and techniques to improve their quality.

Definition 2 A *product-centred modelling technique* is a set of declarative rules that specify the consistent and correct structure of models.

Definition 3 *Product-centred method integration* is the organisation of modelling techniques into a collection within which structural dependencies between different techniques are specified as a set of declarative rules.

The process tradition, in turn, can be called process-centred. It emphasises the structure and organisation of processes with which the products are produced. Methods and techniques introduce explicit methodical procedures that implement the desired or required characteristics of a systems/software development process. They use procedural rules to specify how a correct model should be derived. Methods and techniques do not explicitly specify what kind of model would be correct, but a correct model is what results when the required steps are taken. Method integration combines usage of different techniques aimed at a common goal. In contrast to product-centred methods, process-centred methods combine not only modelling techniques but also other fine-grained tasks. The aim of method integration is to increase the coherence of a systems development process.

The target process of process-centred methods is often called course-grained process, whereas the target of modelling techniques is called fine-grained process. The process can also be called process pattern if the described process can be applied to several methods or techniques.

Definition 4 A *process-centred method* is a method that concentrates on the procedures used in systems development and uses process-centred metrics and techniques to measure and improve their quality.

Definition 5 A *process-centred modelling technique* is a set of modelling steps with which a consistent and correct model is derived.

Definition 6 *Process-centred method integration* is the procedural organisation of the use of modelling techniques and other fine-grained tasks, which specifies how and in which order the techniques are used and tasks performed.

3 Method Engineering and Process Engineering

Method engineering and process engineering are partially overlapping and complementary approaches to method development. The goal of method development is to build up collective experience of systems development and utilise it to craft systematic development practices. Method engineering and process engineering adapt and improve methods for local needs. They aim at increasing accuracy and fitness, and thereby enhancing the feasibility and usefulness of the methods. Their relationship is briefly illustrated in figure 2. Method engineering develops product-centred methods focusing on modelling techniques, whereas process engineering develops process-centred methods focusing on development processes. These techniques and processes are then used in systems development to produce system products.

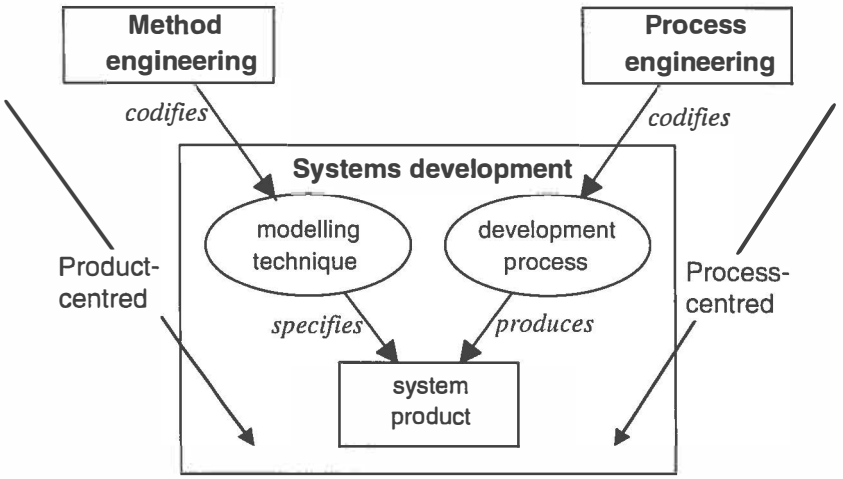

FIGURE 2 Method engineering and process engineering are two approaches to develop methods for systems development.

Method engineering has emerged in the method tradition. This is why its practical view of methods is overwhelmingly product-centred. Method engineering aims to improve accuracy of methods in that techniques would aid to produce consistent and correct models, and fitness in that the techniques would be effective for a particular modelling purpose. It can play two roles in method development: method adaptation and method improvement. Method adaptation focuses on creating and adapting methods for local needs, whereas method improvement concentrates on assessing and calibrating local methods.

It is likely due to the general novelty of method engineering, that it has been considered merely as a means of method adaptation. For long, it was thought that once a method is adapted to an organisation, there is no need for further modifications. Today, the emergence of incremental method engineering is altering this oldish thinking by stressing the role of gradual method evolution (Tolvanen, 1998). However, it is not well known how method

improvement should be facilitated besides providing explicit customisation mechanisms in method support tools. Rossi and Brinkkemper (1996) are the first to propose a systematic approach for measuring properties of methods, but they intend the metrics to support method selection, not improvement.

Definition 7 *Method engineering* is a discipline dedicated to the study, design, construction and adaptation of methods for a specific organisation.

Definition 8 *Method adaptation* is the process of creating and adjusting methods for the needs of a specific organisation.

Definition 9 *Method improvement* is the systematic enhancement of an organisation's methods through local assessment and calibration.

Definition 10 *Method evolution* is the gradual alteration of methods used in a specific organisation.

Process engineering has emerged in the process tradition. It maintains a process-centred view and prefers the term 'process' for denoting a method. Process engineering aims to improve accuracy of processes in that the product produced would match the intended result, and fitness in that people involved would be able to faithfully follow specified actions. It may play two roles in process specification: process adaptation and process improvement. Process adaptation focuses on specifying and adjusting processes according to local needs, whereas process improvement concentrates on assessing process performance and establishing methodical procedures.

Process engineering has been less interested in process adaptation that in process improvement. This is supposedly due to a narrow emphasis on economical efficiency and productivity, which tends to exclude sociological and organisational concerns (Sommerville and Rodden, 1995). Process adaptation is considered solely as a means to adjust technology to process evolution (Madhavji, 1991; Bandinelli and Fuggetta, 1993; Heineman et al., 1994), whereas process improvement has long been a major focus of research. The most salient outcomes of this research are process improvement models such as CMM (Paulk et al., 1993), SPICE (Dorling, 1993), and Bootstrap (Haase et al., 1994).

Definition 11 *Process engineering* is a discipline dedicated to the study, design, implementation and improvement of systems development processes in a specific organisation.

Definition 12 *Process adaptation* is the process of specifying and adjusting development procedures according to the needs of an organisation.

Definition 13 *Process improvement* is the systematic enhancement of an organisation's development processes through local assessment and establishment of methodical procedures.

Definition 14 *Process evolution* is the gradual change in a conducted process in contrast to one prescribed, caused by changes in the requirements or contingencies of the local organisation.

4 Method Modelling

A core function of both method engineering and process engineering is method modelling. Both disciplines largely agree on what aspects a comprehensive method specification should address (Marttiin et al., 1995; Conradi et al., 1992). Method models are also used for similar purposes. Descriptive method models aid in method evaluation, communication and learning, and they should be understandable and easily comparable. Prescriptive method models specify and articulate methods to be used. For automated support, they need to be strict and may involve technology-specific components. Despite these similarities, method modelling yet most reflects the differences between the two engineering approaches.

4.1 Product-Centred Method Modelling

Method engineering has promoted a product-centred view of method modelling. The method model is an integrated collection of metamodels, each of which captures specific information about one type of system models in that method. Examples are class diagrams, use case diagrams, and state chart diagrams in UML (Booch et al., 1999). This information mostly concerns notations and metadata. The type of method modelling used in method engineering is called metamodelling. Metamodelling is carried out using a metamodelling language.

In figure 3, two modelling domains are shown, across which the specification and use of metamodels are organised. These domains are at different abstraction levels (systems development and method engineering) but both are centred on a modelling activity (Brinkkemper, 1990; Tolvanen et al., 1996). First, system modelling is a modelling activity that appears on the lower abstraction level. System modelling perceives a system and develops system models using a method. The method incorporates a set of modelling techniques and it is codified into a set of metamodels. Second, metamodelling is a modelling activity one level higher to system modelling. Metamodelling perceives a method and develops a set of respective metamodels using a metamethod. The metamethod incorporates a metamodelling language and it is codified into a meta-metamodel.

Definition 15 A *metamodel* is a model that specifies a product-centred modelling technique by capturing information about one type of models in a declarative form.

Definition 16 A *metamodelling language* is a modelling language used for representing the declarative rules of a product-centred modelling technique.

Definition 17 *Metamodelling* is the process of specifying a metamodel using a metamodelling language.

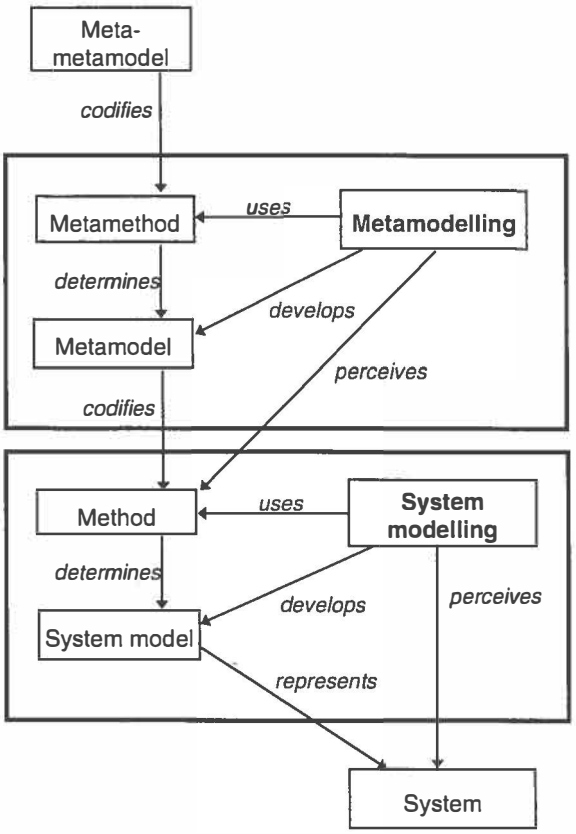

FIGURE 3 Metamodelling is a modelling activity one abstraction level higher to system modelling.

4.2 Process-Centred Method Modelling

Process engineering has promoted a process-centred view of method modelling. The method model used is a process model that captures information about a development process. This information concerns different elements present in a development process. Examples are workers, activities, artefacts, and workflows in RUP (Kruchten, 1998). A process model shows how to produce system products, and how to manage associated development and managerial activities. We can distinguish between generic process templates and project-specific process models. The former is complemented with project-specific information to produce the latter. The type of method modelling used in process engineering is called process modelling and it is carried out using a process modelling language.

In figure 4, two process domains are shown, across which the specification and use of process models are organised. Also these domains are at different abstraction levels (systems development and process engineering) but they are centred on a specific process. First, a production process locates on the lower abstraction level. It develops an information system following a specific method. The method is codified into a process model and it organises the use of modelling techniques. These techniques are used for specification of system models to represent the system under development. Second, a metaprocess

locates one level higher to a production process. It incorporates process elicitation, adaptation, assessment, and improvement. Part of process elicitation is process modelling that specifies the production process in a process model. Metaprocess follows a metamethod codified in a metaprocess model.

Definition 18 A *process model* is a model that specifies a process-centred method by capturing information about a development process.

Definition 19 A *process template* is a generic process model that can be instantiated and supplemented with project-specific data to form a process model for a particular system project.

Definition 20 A *process modelling language* is a modelling language used for representing the procedural rules of a process-centred method.

Definition 21 *Process modelling* is the process of specifying a process model or process template using a process modelling language.

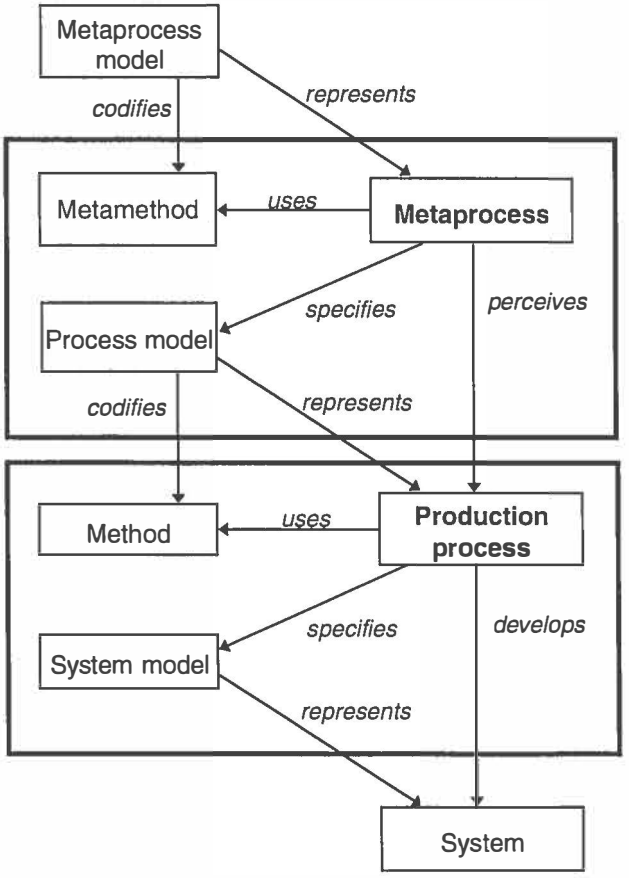

FIGURE 4 Metaprocess is a process one abstraction level higher to a production process.

5 Technology for Method Use and Customisation

When a development group follows the rules and guidelines of a method in systems development, the method use is called method enactment. In product-centred method enactment, a development group uses a set of modelling techniques, whereas in process-centred method enactment it follows a process model. In the latter case, the commonly used term is process enactment.

Definition 22 *Method enactment* is the actual use and conformance to a method in systems development.

Definition 23 *Product-centred method enactment* is the enactment of a product-centred method in systems development by obeying the declarative rules of a metamodel.

Definition 24 *Process enactment* is the enactment of a process-centred method in systems development by following the procedural rules of a process model.

Method enactment can be performed manually or using automated support (McChesney, 1995). First, manual enactment is facilitated through some organisational requirement or contractual obligation to conform to a specific method. A development group refers to method specifications, standards and manuals (e.g., method text-books or process manuals) to use the method correctly. The emphasis of manual enactment is on the actors to conform to the recommended techniques and procedures. Second, automated enactment is facilitated through some degree of automation. The method model used in a method support tool may remain implicit or explicit, depending on whether the method is built into the tool or some automated mechanism is used to enact a method model.

Automated method support is concerned with the level of tool involvement in the development process, that is, how strictly the method rules are implemented. A product-centred approach determines how strictly a tool preserves the consistency and correctness of a product (cf. Vessey et al., 1992). In contrast, a process-centred approach determines how strictly a tool enforces the correctness of the design process (cf. McChesney, 1995). For example, a product-centred ERA (entity-relationship-attribute) technique cannot enforce the user to create entities and attributes in a specific order but instead ensures that the entities and attributes are ultimately connected to each other. In contrast, a process-centred ERA technique could enforce that an attribute is created only when it can (and will) be directly connected to an existing entity.

The importance of automated support is acknowledged in both traditions. Smolander et al. (1990) observe that increased quality in method use seems to flow almost exclusively from the use of tools. Similarly, Stenning (1987) finds that the practicality of a process model may be critically dependent on how much there is automated support for its use. However, tool design is found to highly influence the actual benefits gained from method use. Not only the

method has to suit the development situation, but also the tool has to suit for supporting the method. These are equally important for successful method use.

The way to adapt support tools is customisation. Customisable tools are useful for organisations that are not experienced with methods, since they allow an organisation to assess and evaluate different methods to select the suitable ones. They also suit an organisation when the future method requirements are fuzzy or evolving. Method customisation technology also gives an organisation a possibility to use a method of its own and to create tool support for it. Such technology is especially useful when an organisation is using several methods or local variations of textbook methods (Bubenko, 1988; Brinkkemper, 1990).

5.1 Method Support

Computer Aided Systems/Software Engineering (CASE) is an approach to systems development that involves the use of computer aided tools. It aims to enhance the applicability of methods through standardisation, normalisation and automation. CASE technology comprises mainly production technology focusing on the creation, analysis and transformation of system models (Henderson and Cooprider, 1994). The technology implements automated support for the use of methods in systems development. A CASE tool implements automated support for a specific task, whereas a CASE environment integrates a set of CASE tools that cover several parts of the systems development life-cycle.

Definition 25 *Computer Aided Systems/Software Engineering* is a disciplined approach to systems development, in which computers are used to provide some automated support for the use of methods in systems development.

Definition 26 A *CASE tool* is a design aid tool that implements automated support for one prominent task of systems development.

Definition 27 A *CASE environment* is a collection of CASE tools that cover several parts of the systems development life-cycle.

5.2 Product-Centred Method Support

The method tradition has brought forth two areas of study on product-centred method support technology and its use in systems development. First, the tradition has provided automated tools for the use of product-centred methods and modelling techniques in requirements engineering. A product-centred CASE environment automates product-centred method integration. The core of such a CASE environment architecture is a structured framework that composes and integrates different parts of the environment (ECMA, 1993). A product-centred CASE tool implements the declarative rules of a product-centred modelling technique.

Definition 28 A *product-centred CASE environment* is a CASE environment that supports product-centred method integration.

Definition 29 A *product-centred CASE tool* is a CASE tool that implements and supports the use of a product-centred modelling technique.

Second, the method tradition has introduced method adaptation in CASE. This new area, metaCASE, involves customisation of method support to enhance its applicability and fitness in CASE. A metaCASE environment combines a set of metaCASE tools in the same way as a CASE environment combines CASE tools. They differ in that methods are not fixed into metaCASE tools but the tools are generated or customised for the specific use (Alderson, 1991).

A metaCASE tool is based either on an implicit or explicit metamodel. In the former case, the tool is generated from a metamodel and inserted into a metaCASE environment. In the latter case, a metaCASE tool has a generic architecture that makes it able to adapt to a given metamodel. The metaCASE environment architecture is based on a metaengine that performs all metamodel enactment and handles all access to a common repository (Kelly et al., 1996).

Definition 30 *MetaCASE* is an area of CASE, in which product-centred method support is generated from metamodels.

Definition 31 A *metaCASE tool* is a CASE tool that is generated or customised to support a specific modelling technique.

Definition 32 A *metaCASE environment* is a CASE environment that collects a set of metaCASE tools and includes mechanisms either for metamodel enactment or for inserting generated metaCASE tools into the environment.

Definition 33 A *metaengine* is an automated mechanism that enacts a metamodel to provide tool support for the use of a product-centred modelling technique.

Furthermore, the method tradition has brought forth an area called Computer Aided Method Engineering (CAME). This area studies computer-aided construction and adaptation of methods. It goes to say that CAME is the "CASE" of method engineering. It develops CAME technology – CAME tools and CAME environments – in the same broadness as CASE develops CASE technology.

Definition 34 *Computer Aided Method Engineering* is a disciplined approach to method development, in which computers are used to support or automate some of the tasks.

Definition 35 A *CAME tool* is a design aid tool that implements automated support for one prominent task of method engineering.

Definition 36 A *CAME environment* is a collection of CAME tools that covers several parts of the method engineering life-cycle.

5.3 Process-Centred Method Support

The process tradition has focused on process-centred technologies. First, a process-centred CASE environment automates process-centred method integration by supporting co-ordination of coarse-grained development processes. For example, they may support tool invocation at appropriate times and control resource sharing between system developers. A process-centred CASE tool implements the procedural rules of a process-centred modelling technique and thus supports the fine-grained process of model construction.

Definition 37 A *process-centred CASE environment* is a CASE environment that supports process-centred method integration.

Definition 38 A *process-centred CASE tool* is a CASE tool that implements the procedural rules of a process-centred modelling technique.

Second, the process tradition has brought forth Process Centred Systems/Software Engineering (PCSE) that studies the customisation and use of process-centred methods. It attempts to enhance the applicability of process models through formalisation and automation. A PCSE environment combines a set of CASE tools and integrates their use according to a process model. The core of a PCSE environment architecture is a process engine, which is a mechanism that enacts a process model and evokes guidance and support for the users accordingly.

Definition 39 *Process Centred Systems/Software Engineering* is an area of CASE, in which process-centred method support is generated from process models.

Definition 40 A *PCSE environment* is a CASE environment that supports process-centred method integration according to a process model.

Definition 41 A *process engine* is an automated mechanism that enacts a process model to provide support for system developers accordingly.

Process customisation technology has been studied under the broad umbrella of PCSE. However, it is useful to distinguish between customisable process support technology and process customisation technology the same way as between metaCASE and CAME. We consider PCSE as a counterpart of metaCASE and call the other Computer Aided Process Engineering (CAPE). CAPE is the "CASE" of process engineering, providing it with process engineering method support. Unlike PCSE technology, CAPE technology does not include process support but functions as design aid for process engineers. A CAPE environment may be customisable in the same way as a metaCASE environment.

Definition 42 *Computer Aided Process Engineering* is a disciplined approach to process development, in which computers are used to support or automate some of the tasks.

Definition 43 A *CAPE tool* is a design aid tool used that implements automated support for one prominent task of process engineering.

Definition 44 A *CAPE environment* is a collection of CAPE tools that cover several parts of the process engineering life-cycle.

6 Strategic Integration Points of a Customisable Design Environment

As a result of the study, we outlined a design environment architecture that would address and integrate different aspects discussed above. To develop such an architecture it is first necessary to specify the dependencies and integration mechanisms between its sub-domains. This cannot be done one domain at a time but we need to consider the environment architecture a whole. Without a standard that defines this integration, we cannot safely specify the independent sub-domains.

There are three main sub-domains in this architecture: CASE, CAME and CAPE domains. The two latter should be based on a similar design as the former to enable extensive customisation. Between these domains, we find five strategic points of integration (figure 5).

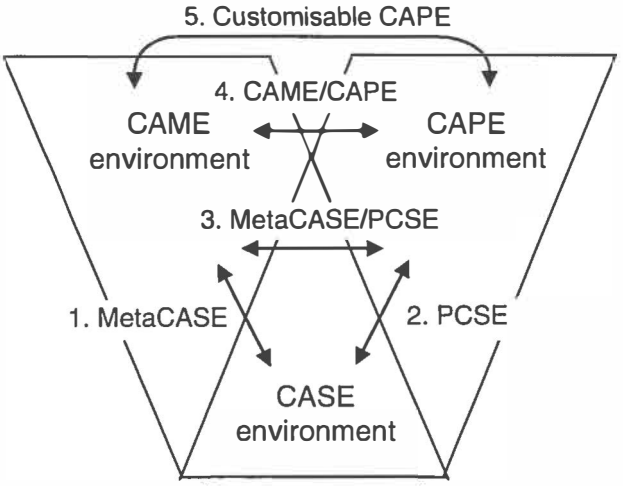

FIGURE 5 Five strategic integration points of a customisable design environment.

1. MetaCASE: Integration of CAME and CASE. First, we have to decide how CASE tools use the products of CAME. An appropriate solution is found in (Kelly, 1996). MetaCASE tools are based on a generic tool architecture that a metaengine complements according to a metamodel. The design allows runtime changes in the metamodel. Such a design could be extended to support adaptable process patterns (Si-Said et al., 1996; Pohl et al., 2000; Lyytinen et al., 1998).

2. PCSE: Integration of CAPE and CASE. Second, we have to decide how CASE tools use the products of CAPE. A PCSE architecture uses a process engine that is able to "read" process models structured according to a specific process metamodel. Some environments also include a mechanism that allows the propagation of run time changes on process models (Bandinelli and Fuggetta, 1993; Jaccheri and Conradi, 1993; Kaiser and Ben-Shaul, 1993). Compared to metaCASE, method evolution is more difficult since a PCSE environment has to maintain also the consistency of process enactment state.

3. Integration of metaCASE and PCSE. Third, we have to decide how metaCASE tools are managed by a PCSE environment. A generic approach to build CASE tools is required to enable communication between coarse-grained and fine-grained processes (Barghouti and Feiler, 1993; Pohl and Jarke, 1992). MetaCASE tools can be designed to allow multiple operational versions of themselves, each of which is added as an entry in its process programming interface. A specification of such a version would determine what operations are available or required while the particular tool version is used. It also specifies the necessary feedback and intervention points for structured communication. While a metaengine is responsible for enacting the operational tool version, a process engine only needs to react on the structured messages it receives from the metaengine. Recursive tool invocations should be avoided by forwarding all further invocations back to the process engine. In this way, the process engine maintains control over the process without disrupting the performance of the metaengine.

4. Integration of CAME and CAPE. Fourthly, we have to decide how CAPE uses the products of CAME. For coarse-grained process support, the interface between metamodels and process models have to be specified. Those metatypes in CAME, the instances of which (i.e., metamodel entities) are directly referenced in tool invocation specifications, should be integrated with process metamodel entities in CAPE, the instances of which (i.e., process model entities) specify tool invocation. For fine-grained process support, the integrated entities should be similarly chosen but at the level of model manipulation. In customisable CAPE, the integration has to be founded at a higher level: between the meta-metatypes in CAME and the process metatypes in CAPE. The interface should be made simple to ease the manageability of method specifications. Moreover, we need mechanisms to track any changes in CAME products that might affect the products of CAPE.

5. Integration of customisable CAPE. Lastly, we have to decide how CAME can be utilised to customise process modelling techniques used in CAPE. In case we want to maintain a possibility for runtime customisation during process engineering, the architecture has to resemble the one in metaCASE. It should contain a metaengine that is able to use customised process metamodels. Further, in case we need runtime customisation during systems development, we need a generic process engine that is able to "read" a process metamodel and to adjust its operation according to it. This entails that we must differentiate between (1) the generic, built-in operational semantics of

a process engine, and (2) the language-specific, not built-in semantics used to guide the process engine.

Clearly, the more customisation is allowed during runtime, the more complex it will be to maintain consistency in a method support environment. Therefore it is necessary to study in each situation to which extent runtime customisation should be allowed. The degree of customisation can be restricted so that the customised environment functions, e.g., as a CASE environment, a metaCASE environment, or a CASE/PCSE environment.

7 Conclusions

In this paper we have proposed definitions for method and process engineering. These areas have grown separately mainly because of different emphasis. Method tradition has focused on systems analysis and design, while process tradition concentrated on software design and implementation. The dominant difference is the view of methods: product-centred vs. process-centred. This difference is a guiding notion to understand other major differences between method engineering and process engineering, in method modelling, and in the technology for method use and customisation.

Yet, the most striking finding in this study is perhaps a substantial failure to consider systems development to comprise an information system of its own. Consequently, the methodical traditions lack insight in how methods relate and contribute to the structure and formation of these 'meta-information systems' and vice versa. 'Metasystems' are regarded as software with which methods are specified and implemented (Osterweil, 1987; Boloix et al., 1991; Chen, 1988; Karrer and Scacchi, 1993). We find that the diversity and variability of meta-information systems are not recognised well enough. A more profound understanding of the nature of methods and the context in which they operate is required.

This study aims at a more comprehensive and balanced view of methods and method research, and thereby contribute to the further integration of the methodical traditions. We have compared the traditions and illustrated their views with numerous definitions. Based on these definitions, we have also suggested five integration points that are strategic for future tool development. We are convinced that a better understanding of the methodical traditions, especially the need for adaptability in methods and technologies, is required to provide better solutions and tools for this new millennium.

References

Aaen, I., Siltanen, A., Sørensen, C. & Tahvanainen, V.-P. 1992. A Tale of Two Countries - CASE Experiences and Expectations. In K.E. Kendall, K. Lyytinen & J.I. DeGross (Eds.) The Impact of Computer Supported Technologies on Information Systems Development. Amsterdam: Elsevier Science Publishers, 61-93.

Alderson, A. 1991. Meta-CASE Technology. In A. Endre & H. Weber (Eds.) Software Development Environments and CASE Technology, LNCS 509. Berlin: Springer-Verlag, 81-91.

Armenise, P., Bandinelli, S., Ghezzi, C. & Morzenti, A. 1993. A survey and assessment of software process representation formalisms. International Journal of Software Engineering And Knowledge Engineering, 3, 3, 410-426.

Auramäki, E., Hirschheim, R. & Lyytinen, K. 1992. Modelling Offices Through Discourse Analysis: A Comparison and Evaluation of SAMPO with OSSAD and ICN. The Computer Journal, 35, 5, pp. 492-500.

Avison, D.E. & Fitzgerald G. 1988. Information Systems Development: Methodologies, Techniques and Tools. Oxford: Blackwell.

Baker, F.T. 1972. Chief Programmer Team Management of Production Programming. IBM Systems Journal, 11, 1, 56-73.

Bandinelli, S. & Fuggetta, A. 1993. Computational Reflection in Software Process Modeling: the SLANG Approach. In Proceedings of the 15th International Conference on Software Engineering. Los Alamitos: IEEE Computer Society Press, 114-154.

Barghouti, N.S. & Feiler, P.H. 1993. Demonstration Experience Report Session summary. In W. Schäfer (Ed.), Proceedings of the 8th International Software Process Workshop: State of the Practice in Process Technology. IEEE Computer Society Press, 2-5.

Bergheim, G., Sandersen, E. & Solvberg, A. 1989. A taxonomy of concepts for the science of information systems. In E.D. Falkenberg & P. Lindgreen (Eds.) Information System Concepts: an In-Depth Analysis. Amsterdam: Elsevier Science Publishers, 269-321.

Boloix, G., Sorenson, P.G. & Tremblay, J.P. 1991. On Transformations Using A Metasystem Approach To Software Development. The University of Alberta, Edmonton. Technical report.

Booch, G, Rumbaugh, J. & Jacobson, I. 1999 The Unified Modeling Language: User Guide. Reading, MA: Addison-Wesley.

Brinkkemper, S. 1990. Formalization of Information Systems Modelling. University of Nijmegen. Nijmegen: Thesis Publishers, Ph.D. Thesis.

Bubenko, J.A. jr. 1988. Selecting a strategy for computer-aided software engineering (CASE). University of Stockholm, SYSLAB Report No 59.

Chen, M. 1988. The Integration of Organization and Information Systems Modeling: A Metasystem Approach to the Generation of Group Decision Support Systems and Computer-Aided Software Engineering. University of Arizona, unpublished Ph.D. Thesis.

Conradi, R., Fernström, C., Fuggetta, A. & Snowdon, R. 1992. Towards a Reference Framework for Process Concepts. In J.-C. Derniame (Ed.) Software Process Technology, EWSPT'92, LNCS 635. Berlin: Springer-Verlag, 3-17.

Conradi, R., Fuggetta, A. & Jaccheri, M.L. 1998. Six Theses on Software Process Research. Software Process Technology, EWSPT'98, LNCS 1487. Berlin: Springer-Verlag, 100-104.

Curtis, B., Kellner, M.I. & Over, J. 1992. Process modeling. Communications of the ACM, 35, 9, 75-90.

Davis, G.B. & Olson, M.H. 1985. Management Information Systems, Conceptual Foundations, Structure and Development. New York: McGraw-Hill.

DeMarco, T. 1996. The Role of Software Development Methodologies: Past, Present, and Future. Proceedings of the 18th International Conference on Software Engineering. Los Alamitos: IEEE Computer Society Press, 2-4.

Dijkstra E. W. 1969. Structured Programming. In J.N. Buxton & B. Randell (Eds) Software Engineering Techniques. Brussels, Belgium: NATO Science Committee.

Dorling, A. 1993. SPICE: Software process improvement and capacity determination. Information and Software Technology, 35, 6/7, 404-406.

Osterweil, L.J. 1987. Software processes are software too. In Proceedings of the 9th International Conference on Software Engineering. Washington D.C.: Computer Society of the IEEE, 12-13.

European Computer Manufactures Association (ECMA). 1993. Reference Model for Frameworks of Software Engineering Environments, ECMA TR/55, NIST Special Publication 500-211.

Haase, V., Messnarz, R., Koch, G., Kugler, H.J. & Decrinis, P. 1994. Bootstrap: Fine-Tuning Process Assessment. IEEE Software, July 1994, 25-35.

Harmsen, F., Brinkkemper, S. & Oei, H. 1994a. Situational Method Engineering for Information System Projects. In T.W. Olle & A.A. Verrijn-Stuart (Eds.) Proceedings of the IFIP WG8.1 Working Conference CRIS'94. Amsterdam: North-Holland Publishers, 169-194.

Heineman, G.T., Botsford, J.E., Caldiera, G., Kaiser, G.E., Kellner, M.I. & Madhavji, N.H. 1994. Emerging technologies that support a software process life cycle. IBM Systems Journal, 33, 3, 501-529.

Henderson, J.C. & Cooprider, J.G. 1994. Dimensions of IS Planning and Design Aids: A Functional Model of CASE Technology. In T. Allen & M. Scott-Morton (Eds.) IT and the Corporation of the 1990's: Research studies. New York: Oxford University Studies Press, 221-248.

Heym, M. & Österle, H. 1993. Computer-aided methodology engineering. Information and Software Technology, 35, 6/7, 345-354.

Hirschheim, R., Klein, H. & Lyytinen, K. 1995. Information Systems Development: Conceptual and Philosophical Foundations. Cambridge: Cambridge University Press.

ISO/IEC. 1990. ISO/IEC 10027 Information Technology - Information Resource Dictionary System (IRDS) - Framework. ISO/IEC International Standard.

Jaccheri, M.L. & Conradi, R. 1993. Techniques for Process Model Evolution in EPOS. IEEE Transactions on Software Engineering, 19, 12, 1145-1156.

Jarke, M., Pohl, K., Rolland, C. & Schmitt, J.-R. 1994. Experience-Based Method Evaluation and Improvement: A process modeling approach. In T.W. Olle & A.A. Verrijn-Stuart (Eds.) Proceedings of the IFIP WG8.1 Working Conference CRIS´94. Amsterdam: North-Holland, 1-27.

Kaiser, G.E. & Ben-Shaul, I.Z. 1993. Process Evolution in the Marvel Environment. In W. Schaefer (Ed.) Proceedings of the 8th International Software Process Workshop. Los Alamitos: IEEE Computer Society Press, 104-106.

Karrer, A.S. & Scacchi, W. 1993. Meta-environments for software production. International Journal of Software Engineering and Knowledge Engineering, 3, 1, 139-162.

Kelly, S., Lyytinen, K. & Rossi, M. 1996. METAEDIT+ — A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In P. Constantopoulos, J. Mylopoulos & Y. Vassiliou (Eds.) Advanced Information Systems Engineering, LNCS 1080. Berlin: Springer-Verlag, 1-21.

Koskinen, M. & Marttiin, P. 1998. Developing a Customisable Process Modelling Environment: Lessons Learnt and Future Prospects. In V. Gruhn (Ed.) Proceedings on the 6th European Workshop on Software Process Technology, EWSPT'98, LNCS 1487. Springer-Verlag, 13-27.

Kruchten, P. 1998. The Rational Unified Process: An Introduction. 2nd edition. Reading, MA: Addison-Wesley.

Lonchamp, J. 1993. A structured conceptual and terminological framework for software process engineering. In L. Osterweil (Ed.) Proceedings of the 2nd International Conference on the Software Process. Los Alamitos: IEEE Computer Society Press, 41-53.

Lyytinen, K., Marttiin, P., Tolvanen, J.-P., Jarke, M., Pohl, K. & Weidenhaupt, K. 1998. Bridging the Islands of Automation. In: S.T. March & J. Bubenko Jr. (eds) Proceedings of the Eight Annual Workshop on Information Technologies and Systems (WITS'98). University of Jyväskylä. Computer Science and Information System Reports, Technical Reports TR-19.

Madhavji, N.H. 1991. The process cycle. Software Engineering Journal, 6, 5, 234-242.

Marttiin, P. 1998. Customisable Process Modelling Support and Tools for Design Environment. University of Jyväskylä. Jyväskylä Studies in Computer Science, Economics and Statistics 43. PhD Thesis.

Marttiin, P., Lyytinen, K., Rossi M., Tahvanainen V.-P., Smolander K. & Tolvanen, J.-P. 1995. Modeling Requirements for Future CASE: modeling issues and architectural considerations. Information Resource Management Journal, 8, 1, 15-25.

McChesney, I.R. 1995. Toward a classification scheme for software process modeling approaches. Information and Software Technology, 37, 7, 363-374.

Olle, T.W., Hagelstein, J., MacDonald, I.G., Rolland, C., Sol, H.G., Van Assche, F.J.M. & Verrijn-Stuart, A.A. 1991. Information Systems Methodologies — A framework for understanding. Wokingham: Addison-Wesley.

Osterweil, L.J. 1987. Software processes are software too. In Proceedings of the 9th International Conference on Software Engineering. Washington D.C.: Computer Society of the IEEE, 12-13.

Paulk, M.C., Curtis, B., Chrissis, M.B. & Weber, C.V. 1993. The Capability Maturity Model: Version 1.1. IEEE Software, July, 18-27.

Pohl, K. 1996. Process-Centered Requirements Engineering. New York: Wiley.

Pohl, K. & Jarke, M. 1992. Quality Information Systems: Repository Support for Evolving Process Models. Aachener Informatik-Berichte 92-37, RWTH Aachen, Germany.

Pohl, K., Weidenhaupt, K. Dömges, R., Haumer, P., Jarke, M., & Klamma, R. 2000. PRIME – Toward Process Integrated Modeling Environments. ACM Transactions on Software Engineering and Methodology, 8, 4, 343-410.

Purper, C.B. 2000. Transcribing Process Model Standards into Meta-Processes. In: R. Conradi (ed.) Software Process Technology, EWSPT 2000. LNCS 1780. Berlin: Springer-Verlag, pp. 55-68.

Pyburn, P. 1983. Linking the MIS Plan with Corporate Strategy: An Exploratory Study. MIS Quarterly, June, 1-14.

Rolland, C., Souveyet, C. & Moreno, M. 1995. An approach of defining ways-of-working. Information Systems, 20, 4, 337-359.

Rossi, M. & Brinkkemper, S. 1996. Complexity Metrics for Systems Development Methods and Techniques. Information Systems, 21, 2, 209-227.

Royce, W.W. 1970. Managing the Development of Large Software Systems. In Proceedings Wescon, New York: IEEE Computer Society Press, 1-9. (Reprinted in Proceedings of the 9th International Conference on Software Engineering. IEEE Computer Society Press, 1987, 328-338.)

Si-Said, S., Rolland, C. & Grosz, G. 1996. MENTOR: A Computer Aided Requirements Engineering Environment. In P. Constantopoulos, J. Mylopoulos & Y. Vassiliou (Eds.) Advanced Information Systems Engineering, LNCS 1080. Berlin: Springer-Verlag, 22-43.

Smolander, K., Tahvanainen, V.-P. & Lyytinen, K. 1990. How to Combine Tools and Methods in Practice — a Field Study. In B. Steinholtz, A. Sølvberg & L. Bergman (Eds.) Advanced Information Systems Engineering LNCS 436. Berlin: Springer-Verlag, 195-211.

Sommerville, I. & Rodden, T. 1995. Human, Social and Organisational Influences on the Software Process. Technical Report GSEG/2/1995, Computing Department, Lancaster University, UK.

Stenning, V. 1987. On the Role of an Environment. In Proceedings of the 9th International Conference on Software Engineering. Los Alamitos: IEEE Computer Society Press, 30-34.

Teichroew, D. & Hershey III, E.A. 1977. PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems. IEEE Transactions on Software Engineering, 3, 1, 41-48.

Tolvanen, J.-P. 1998. Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence. University of Jyväskylä. Jyväskylä Studies in Computer Science, Economics and Statistics 47. Ph.D. Thesis

Tolvanen, J.-P., Rossi, M. & Liu, H. 1996. Method Engineering: Current research directions and implications for future research. In S. Brinkkemper, K. Lyytinen & R.J. Welke (Eds.) Method Engineering: Principles of method construction and tool support. London: Chapman & Hall, 296-317.

Vessey, I., Jarvenpaa, S. & Tractinsky, N. 1992. Evaluation of Vendor Products: CASE Tools as Methodology Companions. Communications of the ACM, 35, 4, 90-105.

Waters, S.J. 1974. Computer-aided Methodology of Computer Systems Design. The Computer Journal, 17, 3, 211-215.

Wijers, G. 1991. Modeling Support in Information Systems Development. Amsterdam: Thesis publishers, Ph.D. Thesis.

Wijers, G. & van Dort, H. 1990. Experiences with the use of CASE tools in the Netherlands. In B. Steinholz, A. Sølvberg & B. Bergman (Eds.) Advanced Information Systems Engineering, LNCS 436. Berlin: Springer-Verlag, 5-20.

Wynekoop, J.D. & Russo, N.L. 1993. System development methodologies: unanswered questions and the research-practice gap. In J.I DeGross, R.P Bostrom & D. Robey (Eds.) Proceedings of the 14th International Conference on Information Systems. ACM Society Press, 181-190.

Yourdon, E. 1989. Modern Structured Analysis. London: Prentice-Hall.

Yourdon, E. & Constantine, L. 1979. Structured Design. Englewood Cliffs, NJ: Prentice Hall.

PART II: THEORY

3 TOWARDS CUSTOMISATION OF PROCESS MODELLING LANGUAGES IN COMPUTER AIDED PROCESS ENGINEERING

Koskinen, M. "Towards Customisation of Process Modelling Languages in Computer Aided Process Engineering".

The paper has been submitted to the 23rd International Conference on Software Engineering, Toronto, Canada, May 2001.

© 2000 IEEE. Printed with permission.

Towards Customisation of Process Modelling Languages in Computer Aided Process Engineering

Minna Koskinen
University of Jyväskylä

Abstract

Computer Aided Process Engineering (CAPE) technology is used for supporting methodical process engineering. PML customisation for CAPE technology is an emerging area of research that focuses on the specification of process modelling languages and the use of such specifications in process modelling and process enactment. This study investigates PML customisation and outlines its direction in process engineering and related technological support. We find that PML customisation is almost ignored in current research on process engineering. However, there are some empirical studies that consider such customisation essential in local process improvement efforts, since it allows process modelling languages be adapted and evolved according to the local conditions and requirements. Beyond mere linguistic adaptation, PML customisation is almost ignored also in technological research. Instead, we find that current technologies are a major obstacle in the application of PML customisation. In the future, we expect customisable PCSE and CAPE environments to emerge that give organisations more flexibility in process engineering and process improvement.

1 Introduction

Process engineering is a disciplined approach to improve systems development, in which process modelling or process programming is used to articulate how systems development should be carried out. Process research has brought forth a range of process technologies for (mainly software) development processes and process engineering. First, process support technology accounts for the support of conducting methodical systems and software engineering processes. Second, Process-Centred Software Engineering (PCSE) technology is customisable process support technology that incorporates some support for process modelling or process programming. Third, Computer Aided Process Engineering (CAPE) technology is dedicated to methodical process engineering. CAPE technology does not (necessarily) include mechanisms for process support but it provides a set of design aid tools for process engineers.

The change and evolution of processes is recognised as an important issue in process research (Madhavji, 1991; Dowson and Fernström, 1994). For example, Madhavji (1992) presents a general framework for changes that is used as an infrastructure support, within which changes can be carried out by following one or more change methods. The need for customisation is a core reason for the emergence of PCSE and CAPE environments (Bandinelli et al., 1993; Conradi and Jaccheri, 1993; Finkelstein et al., 1994; Kaiser and Ben-Shaul, 1993). First, process evolution refers to changes in the structure of enacting processes. It is extensively discussed in current research literature (Madhavji, 1991; Conradi et al., 1994; Heineman et al., 1994; Lonchamp, 1995). Second, linguistic adaptation refers to changes made to the way in which processes are conceived, represented, or enacted. There are currently few approaches for linguistic adaptation (Balzer and Narayanaswamy, 1993; Kaiser et al., 1996).

Research on the customisation of CAPE technology has emerged only recently. First, metaprocess customisation refers to changes made in process engineering processes. Broadly speaking, the same mechanisms that are used for production process support can be applied also for metaprocess support (Lonchamp, 1995). Second, operational customisation requires mechanisms to change the way in which process model components are created, managed and manipulated. Operational customisation adjusts process modelling tools to different methods of specification, component reuse, and tracing, as examples. Third, language customisation denotes that a process modelling language is adapted in some way (Koskinen, 1999). The changes may concern the conceptual framework, notation or semantics of the language. While the effects of metaprocess and operational customisation are restricted to the use of a CAPE environment, language customisation is instead reflected also as linguistic adaptation in PCSE.

This study investigates PML customisation and outlines its direction in process engineering and related technological support. We shortly review the current state of art in linguistic adaptation and the approaches developed for such adaptation in current PCSE/CAPE technology (in Section 2). Thereafter, we discuss different approaches to process modelling language design and explicate the concept of PML customisation. In addition, we consider some aspects of a next-generation language design required in PML customisation (in Section 3). We also discuss PML customisation in the context of PML engineering and process engineering, and consider the state of art in current process technology (in Section 4). The conclusions are finally briefed (in Section 5).

2 State of Art in Linguistic Adaptation

Linguistic adaptation denotes that some changes can be made to a PCSE environment that affect the way of representing, conceiving, or enacting

processes. Linguistic adaptation may be achieved by several means. We find five general mechanisms of linguistic adaptation in PCSE/CAPE environments.

First, *specialisation* is a mechanism that allows ontological refinement. A PCSE environment that supports specialisation provides a generic process metamodel that contains a set of generic process types or classes. These can be further specialised into process-specific types or classes that possess some additional or more specific properties. The environment thus allows the specification of new process types or classes within a particular process ontology. Specialisation seems to be the most commonly used mechanism of linguistic adaptation in current PCSE/CAPE environments.

As examples, we discuss two environments that use specialisation as the prime mechanism of adaptation. First, Malone et al. (1995) introduce *Oval* that is a tailorable tool for creating co-operative applications. It provides four primitive "building blocks" (objects, views, agents, and links). Environment specific building blocks are created from these primitive blocks through specialisation. Rule-based agents can perform simple tasks triggered by events, e.g., arrival of mail. The tasks are constructed from a set of system-defined generic actions that can operate on different building blocks. Second, Lonchamp (1995) presents *CPCE* that is an environment kernel for managing asynchronous collaborative applications. A process model is a network of classes that are specialised from the generic classes provided by the kernel. Generic classes embody the enactment mechanisms and their generic behaviours are inherited by specialised classes. Class variables describe model properties in a declarative way. Class methods of the generic classes can cope with the anticipated values of these variables, whereas unanticipated resolution methods need a new piece of code included within the kernel.

Second, *metamodelling* collects information common to a class of objects. Through metamodelling, a metamodel is specified that is used as the basis of the integration of a PCSE environment. The concepts specified in a meta-metamodel are instantiated and composed into a metamodel. This metamodel is further instantiated into a specific process model. Specialisation may be used in the construction of type hierarchies.

ConceptBase is a deductive object manager for specification management applications based on a Telos knowledge base (Jeusfeld et al., 1993; Jarke et al., 1993). Jarke and Rose (1992) present the CAD^{O} model that is specified using the O-Telos metamodelling language. The model is used as a metamodel in the further modelling of a software engineering environment. The environment is specified by enumerating the allowed tools, their operations, and the object types processed by the operations.

Third, *generation* takes a process model developed in a CAPE environment and translates it into a process language for which there is a process engine available. The process metamodel of the CAPE environment is first mapped to the target language. This mapping specifies how the translation is carried out. The approach then requires a generator mechanism that implements the mapping.

Again, we consider two examples. First, *Articulator* is a knowledge-based CAPE environment for modelling, analysing, and simulating software processes (Mi and Scacchi, 1990; Mi and Scacchi, 1996). Articulator uses specialisation for the specification of process ontologies and translation for mapping the process models into a process modelling language for which there already exists a process engine. Articulator is implemented on a knowledge-based metamodel with a resource-centred ontology. Each resource model specialises this generic resource model. A special-purpose application generator must be built for the mapping between a specialised class of resource models and a process programming notation. Process models are then automatically transformed into process programs using this generator. Second, *MetaView* is a metaCASE environment for software engineering (Sorenson et al., 1988; Froelich, 1994). MetaView allows the use of different process modelling languages. The system provides an Execution Engine and an intermediary language that is based upon a nested transaction model. Each process modelling language is mapped to this language. At the user level, a particular process modelling language is chosen and a process model or process program is produced with it. The process model is then transformed into an execution model that is interpreted by the Execution Engine.

Fourthly, *delegation* forwards part of the enactment task to external enactment mechanisms. The delegator may be a process engine or other mechanism that manages the core enactment process. There is a mediating exchange interface between the core mechanism and the external mechanism, which makes it possible to co-ordinate their functioning.

PRIME is a framework for constructing process-integrated tools (Pohl et al., 2000). It contains a generic process engine framework that allows the specification of process fragments ("contexts") with different process modelling languages (e.g., SLANG) or programming languages (e.g., C++). The corresponding enactment mechanism has to be embedded in the generic process engine framework. The design of the framework requires that the enactment mechanism provides appropriate interface for message exchange. Also, the suitable process modelling languages are necessarily constrained to those that are compatible with PRIME's context-based process metamodel.

Fifthly, *parameterisation* uses a generic process engine architecture that enables the use of parameters in order to accept extensions and changes to its default enactment semantics. The additions and changes are reflected as notational extensions in the process language. A generic process engine architecture is first proposed by Balzer and Narayanaswamy (1993).

Amber is an extensible rule-based process server (Kaiser et al. 1996). The process assembly language comprises an object-oriented data definition language for the specification of process state and product artefact classes, and a rule language for the specification of actions to be taken by a user or the environment. Notational extensions are made by means of rule annotations. These are strings that can be attached to different sections of the textual rule definitions. Rule annotations affect the default behaviour before, during, or after the execution of rule-sections and the default chaining behaviour into or

from a rule. Semantic extensions are made at specific entry points between rule phases in the interpreter. Variation is permitted both in the interpretation of the rule notation and the semantics of chaining among rules.

The mechanisms of linguistic adaptation supported by current PCSE and CAPE technologies account for up to three forms of adaptation. First, specialisation is the primary means of conceptual adaptation. Specialisation of generic process types introduces new concepts within the conceptual limits of the generic model. This approach is used in most customisable PCSE/CAPE environments. Another approach to conceptual adaptation is metamodelling. Second, notational adaptation is achieved as a "side effect" in parameterisation, in which new keywords are added as notational annotations to reflect linguistic extensions in the process engine. Also the alternative notations brought in by alternative languages in delegation could be seen as some kind of notational adaptation. Third, semantic adaptation is achieved through parameterisation, delegation and generation. Parameterisation allows one to extend the process engine, whereas delegation is based on the use of a mediating interface, and generation on a mapping between the modelling system and a language.

Mechanisms that enable linguistic adaptation are usually not specifically designed with linguistic adaptation in mind but they are more a side-effect of an attempt to achieve other objectives. Specialisation of process types is needed to define process fragments that can be enacted using one generic process interpreter. Metamodelling is used for creating a specific process metamodel that enables a support environment to be consistently integrated. Generation allows the use of a comprehensive CAPE environment for process modelling, analysis, and simulation. Delegation allows the use of different, existing process modelling languages for the specification of process fragments. Parameterisation is the only approach that is genuinely and specifically intended for linguistic adaptation. It makes extensions to the enactment mechanism and changes its default behaviours.

We find that language customisation is hardly considered in current research on process engineering. However, customisable CAPE demands an approach that reaches beyond linguistic adaptation. In the following, we highlight the integral and indispensable role of language customisation in process engineering and thereby also in the application of PCSE and CAPE technologies.

3 PML Customisation

Research on process modelling languages and approaches has been intensive, including both theoretical work on definitions and classifications and experimental work on language design and implementation.

In current classifications, the most frequently addressed aspect of process modelling languages is the "language type", "paradigm" or "style" (Curtis, Kellner and Over 1992; Conradi, Liu and Jaccheri 1991; Madhavji 1991;

Armenise et al. 1993; McChesney 1995). It is considered as the main determinant of the constructs available in a process modelling language. Heterogeneous approaches constitute the first generation of process modelling languages, in which the language characteristics are closely related to the underlying implementation approach – the "base language" (Curtis, Kellner and Over 1992).

Several general language designs have been developed that allow linguistic variation in process modelling. These include a single, semantically broad language, a set of independent and special-purpose languages, and a common core language that can be extended (Sutton, Tarr and Osterweil 1995; Conradi and Liu 1995). The first design, a broad language, provides a broad range of alternative language constructs, from which a process modeller can choose the appropriate ones. The second design, a set of independent languages, provides different languages for expressing different situations or aspects of processes. The third design, an extendible core language, provides a language for expressing the core aspects of processes and allows the extension of modelling system with whatever other language constructs needed.

It is found that a next-generation process modelling language should enable modelling of processes by composing elements from different language paradigms, or representing different semantic aspects (Sutton and Osterweil 1997). This would introduce additional flexibility and incrementality into process modelling. We find that this proposal has two important consequences for language design. First, any particularities of process modelling languages must be abstracted away from language architectures. It has to view language as an artefact with a specific structure, but with no specific content. Second, a further consequence of the first is that the language architecture must prevent the characteristics of a "base language" from reflecting on specific process modelling languages. The generic language architecture must be implementation independent.

PML customisation is the adaptation and evolution of process modelling languages. A process modelling language covers a conceptual framework for the composition, a notation for the representation, and semantics for the interpretation of process models (Koskinen 1999; cf. Lonchamp 1993). Process modelling languages vary in these three aspects. A conceptual framework accounts for a certain perspective of a process and different notations are useful for specific purposes. Semantics determines the rules of interpretation. In full-fledged customisation, both the conceptual framework, notation and semantics of a process modelling language can be specified and adapted.

A PML specification has therefore to distinguish between the conceptual, notational and semantic aspects of a process modelling language. This distinction is beneficial also at the level of process models, since it allows one to maintain several representations of a model and to reuse model components across different process models or process perspectives. Without this representation independence, it would be very difficult to integrate different representations and perspectives (Curtis, Kellner and Over 1992; Sommerville et al. 1995) and especially to maintain the enactment state of process models throughout process changes.

Consequently, we distinguish three classes of language constructs. In the following, we illustrate these classes with an example. The Visual Process Language (VPL) proposed by Shepard, Sibbald and Wortley (1992) is chosen since the article illustrates the language adequately from all three aspects. First, *conceptual constructs* specify process ontologies. VPL provides nine types of concepts: *Start, Finish, Procedure, Task, Decompose, Recompose, Split, Merge,* and *Branch*. A *VPL model* combines a group of different process elements through *Paths*. A *Path* refers to a dependency between two concepts. There are also several constraints that specify how different process elements can and must be conceptually interrelated (e.g., all concepts in a model must be connected by *Paths*). In addition, *Object* is a concept referring to a software artefact.

Second, *notational constructs* are used to implement different representations. A *VPL model* is a directed *Graph* of *Nodes* and *Edges*. *Nodes* represent various concepts while *Edges* represent *Paths*. There are also several notational rules that must apply. Some of these rules are related to the corresponding conceptual specification (e.g., a rule that all the nine types of concept must be represented). Other rules are purely notational (e.g., *Edges* are directed from left to right so that the arrow head points to the node on the right hand side). Some concepts, such as *Object*, have no representation in *Graphs*.

Third, *semantic constructs* specify the rules of model interpretation and enactment. Each semantic construct collects a set of generic enactment features and generic operations. Note that, due to a lack of conceptual differentiation between different construct types, constructs are often labelled with the same name. We use "*(S)*" to distinguish a semantic construct. The enaction of a *VPL program* is the *Flow* of *Objects(S)* through it. This reflects a Petri-net based implementation. A semantic construct determines how this *Flow* proceeds and what happens when certain *Node(S)* is encountered during enactment. The enactment of a *VPL program* begins at *Start(S)* and ends at *Finish(S)*. *Objects(S)* are created at *Start(S)* and deactivated and archived at *Finish(S)*. At *Task(S)*, an *Action* is performed on the *Object(S)*, whereas at *Procedure(S)*, the thread of execution is channelled to a Sub-*Graph(S)*. A *Family of objects* is created at *Decompose(S)* and cumulated and synthesised at *Recompose(S)*. *Split(S)* creates copies of an *Object(S)* and emits one along each output *Flow*, whereas *Merge(S)* is a rendezvous point for the concurrent *Flows* created at *Split(S)*. At *Branch(S)*, one output *Flow* is chosen and an *Object(S)* is emitted along it.

In a PML specification, the different types of language constructs are specified and integrated. The role of PML customisation in process engineering is to create and adapt such specifications.

4 Towards PML Engineering

Process modelling language (PML) engineering is a disciplined approach to the study, design, construction and adaptation of process modelling languages. (Rossi and Sillander 1998). PML customisation is a core function of PML engineering in customisable CAPE.

Many researchers and practitioners have found that the most difficult practical problems of process engineering are not technical. Systematic enhancement of an organisation's processes requires the support of human process actors. Truly successful improvement cannot happen unless the process improvement effort enables the improvement of human awareness, articulation, discussion and negotiation, and above all a change of behaviour. Automation through technology is useful and necessary to the degree it can feasibly support this.

Step-wise improvement is a fundamental notion in process maturity models (Paulk et al. 1993). CMM describes five maturity levels from initial to optimising through which an organisation should mature. However, we find it too strong to presume that an organisation can really be located at a particular maturity level. We often find it more useful to think that an organisation has a *maturity profile* with different weaknesses and strengths. Concerning one aspect an organisation's maturity may be quite low, while regarding another it may be rather high. The first task in process engineering should be to find out this profile and then to adjust the process improvement effort accordingly. A feasible starting point for improvement is to account for the local practices and problems. After that, processes and the respective support should be improved gradually at a speed an organisation is able and prepared to mature. PML engineering should be an organic part of this effort. Process modelling languages should be adapted and evolved according to the given local conditions and requirements.

Rossi and Sillander (1998) report a process engineering effort that included PML engineering as part of it. The effort constituted a four-step process modelling life-cycle conducted by two quality engineers. The life-cycle started with a process context study using data collection. The quality engineers used multiple data sources including semi-formal interviews, questionnaires, observation and participative sessions, local manuals and other documents, and research literature. They also participated in actual software engineering to get an "inside view" of the target process. The second step included PML selection and adaptation with an objective to achieve a process modelling language that would be appropriate for the present modelling situation. The quality engineers surveyed and evaluated existing PMLs based on their understanding of the process context and the forthcoming modelling task. Thereafter, they selected one language that appeared most suitable and further modified it to properly fit their objectives. In the third step, the quality engineers used the adapted PML to create a preliminary process model. This "pilot model" represented the process as the quality engineers understood it.

The PML was further modified where necessary. Lastly, the quality engineers arranged a participative modelling session. During this session, the pilot model was made into a large wall chart that the process actors collectively commented and revised according to their specific needs. Also the PML was slightly modified.

With PML engineering, the usefulness of a process modelling language in a local context is optimised. Therewith, the language fits the context, given purpose and the unique composition of objectives. The constructs it provides for representing and reasoning about various aspects is chosen accordingly. The features of the process modelling language thus become such that are needed and used in real. Moreover, when extensive and participant-intensive data analysis and modelling are used, language features become well understood both among process engineers and process actors.

Unfortunately, the current state-of-art in process technology is a major obstacle in the application of PML engineering. An organisation can obtain process support only by adopting a process support environment together with a particular process modelling language(s) or by developing a process support environment of its own. The latter choice is often out of question due to the high cost of local development. Therefore, an organisation is forced to select a certain technology and approach too early. The organisation is forced to choose process modelling methods and languages and to make decisions on the process support paradigm and tools before an opportunity to properly experience in their use and possible side-effects. In addition, further adaptation and improvement is obstructed unless the invested technology is replaced with another.

PML customisation for CAPE technology is an emerging area of research that focuses on the specification of process modelling languages and the use of such specifications in process modelling and process enactment. It requires a customisable PCSE/CAPE environment that uses explicit PML specifications with a generic process engine, and provides some facilities for PML engineering. An example towards this kind of approach is an environment called CPME that is still in a prototypical stage (Koskinen and Marttiin 1998). In CPME, Process modelling languages are specified through process metamodelling that allows the customisation of process modelling languages (Koskinen and Marttiin 1997). In the future, we expect this type of customisable PCSE and CAPE environments to emerge and to give organisations more flexibility in process engineering and process improvement with lower risks in process modelling and technology adoption.

5 Conclusions

We have studied PML customisation in the context of PML engineering and process engineering. We find that PML customisation is almost ignored in current research on process engineering, despite the few empirical studies that

further consider it essential in local application of process engineering. We find that PML engineering is an integral and often indispensable part of process engineering. Therewith, process modelling languages can be adapted and evolved according to the given local conditions and requirements.

Current PCSE and CAPE technologies provide some means for linguistic adaptation but full-fledged PML customisation is not supported. Furthermore, instead of supporting PML customisation, current technology is a major obstacle in its application. In the future, we expect customisable PCSE/CAPE environments to emerge that give organisations more flexibility in process engineering and process improvement.

References

Armenise, P., Bandinelli, S., Ghezzi, C. & Morzenti, A. 1993. A survey and assessment of software process representation formalisms. International Journal of Software Engineering And Knowledge Engineering, 3, 3, 410-426.

Balzer, R. & Narayanaswamy, K. 1993. Mechanisms for generic process support. In D. Notkin (Ed.) Proceedings of the 1st ACM SIGSOFT Symposium on the Foundations of Software Engineering. Special Issue of Software Engineering Notes, 18, 5, 21-32.

Bandinelli, S., Fuggetta, A. & Ghezzi, C. 1993. Software Process Model Evolution in the SPADE Environment. IEEE Transactions on Software Engineering, 19, 12, 1128-1144.

Conradi, R., Fernström, C. & Fuggetta, A. 1994. Concepts for Evolving Software Processes. In A. Finkelstein, J. Kramer & B. Nuseibeh (Eds.) Software Process Modelling and Technology. New York: Wiley.

Conradi, R. & Jacchieri, M.L. 1993. Customization and Evolution of Process Models in EPOS. In N. Prakash, C. Rolland & B. Pernici (Eds.) Information System Development Process. Amsterdam: Elsevier Science Publishers, 23-39.

Conradi, R. & Liu, Ch. 1995. Process Modelling Languages: One or Many? In W. Schäfer (Ed.) Software Process Technology, EWSPT'95, LNCS 913. Berlin: Springer-Verlag, 98-118.

Conradi, R., Liu, C. & Jaccheri, M.L. 1992. Process modelling paradigms: an evaluation. In M. I. Thomas (Ed.) Proceedings of the 7th International Software Process Workshop. Los Alamitos: IEEE Computer Society Press, 51-53.

Curtis, B., Kellner, M.I. & Over, J. 1992. Process modeling. Communications of the ACM, 35, 9, 75-90.

Deiters, W. & Gruhn, V. 1994. The Funsoft Net Approach to Software Process Management. International Journal of Software Engineering and Knowledge Engineering, 4, 2, 229-256.

Dowson, M. & Fernström, C. 1994. Towards Requirements for Enactment Mechanisms. In B. Warboys (Ed.) Software Process Technology, EWSPT'94, LNCS 772. Berlin: Springer-Verlag, 90-106.

Finkelstein, A., Kramer, J. & Nuseibeh, B. 1994. Software Process Modelling and Technology. New York: Wiley.

Froehlich, G. 1994. Process Modeling Support in Metaview. Department of Computational Science, University of Saskatchewan, Saskatchewan, Canada. Master's Thesis.

Heineman, G.T., Botsford, J.E., Caldiera, G., Kaiser, G.E., Kellner, M.I. & Madhavji, N.H. 1994. Emerging technologies that support a software process life cycle. IBM Systems Journal, 33, 3, 501-529.

Jarke, M., Jeusfeld, M. & Rose, T. 1993. Process Services in ConceptBase. In M. Jarke (Ed.) Database Application Engineering with DAIDA. Berlin: Springer-Verlag, 389-412.

Jarke, M. & Rose, T. 1992. Specification Management with CAD°. In P. Loucopoulos & R. Zicari (Eds.) Conceptual Modeling, Databases, and CASE. New York: Wiley, 489-505.

Jeusfeld, M., Rose, T. & Jarke, M. 1993. ConceptBase: A Telos-Based Software Information System. In M. Jarke (Ed.) Database Application Engineering with DAIDA. Berlin: Springer-Verlag, 367-388.

Kaiser, G.E. & Ben-Shaul, I.Z. 1993. Process Evolution in the Marvel Environment. In W. Schaefer (Ed.) Proceedings of the 8th International Software Process Workshop. Los Alamitos: IEEE Computer Society Press, 104-106.

Kaiser, G.E., Ben-Shaul, I.Z., Popovich, S.S. & Dossick, S.E. 1996. A Metalinguistic Approach to Process Enactment Extensibility. In W. Schaefer (Ed.) Proceedings of the 4th International Conference on the Software Process. Los Alamitos: IEEE Computer Society Press, 90-101.

Koskinen, M. 1999. A Metamodelling Approach to Process Concept Customisation and Enactability in MetaCASE. University of Jyväskylä. Computer Science and Information Systems Reports, Technical Reports TR-20. Licentiate thesis.

Koskinen, M. & Marttiin, P. 1997. Process Support in MetaCASE: Implementing the Conceptual Basis for Enactable Process Models in MetaEdit+. In J. Ebert & C. Lewerentz (Eds.) Software Engineering Environments. Los Alamitos: IEEE Computer Society Press, 110-123.

Koskinen, M. & Marttiin, P. 1998. Developing a Customisable Process Modelling Environment: Lessons Learnt and Future Prospects. In V. Gruhn (Ed.) Proceedings on the 6th European Workshop on Software Process Technology, EWSPT'98, LNCS 1487. Springer-Verlag, 13-27.

Lonchamp, J. 1995. CPCE: A Kernel for Building Flexible Collaborative Process-Centered Environments. In M.S. Verrall (Ed.) Software Engineering Environments. Los Alamitos: IEEE Computer Society Press, 28-41.

Madhavji, N.H. 1991. The process cycle. Software Engineering Journal, 6, 5, 234-242.

Madhavji, N.H. 1992. Environment Evolution: The Prism Model of Changes. IEEE Transactions on Software Engineering, 18, 5, 380-392.

Malone, T.W., Lai, K.-Y. & Fry, C. 1995. Experiments with Oval: A Radically Tailorable Tool for Cooperative Work. ACM Transactions on Information Systems, 13, 2, 177-205.

McChesney, I.R. 1995. Toward a classification scheme for software process modeling approaches. Information and Software Technology, 37, 7, 363-374.

Mi, P. & Scacchi, W. 1990. A Knowledge-Based Environment for Modeling and Simulating Software Engineering Processes. IEEE Transactions on Knowledge and Data Engineering, 2, 3, 283-294.

Mi, P. & Scacchi, W. 1996. A Meta-Model for Formulating Knowledge-Based Models of Software Development. Decision Support Systems, 17, 3, 313-330.

Paulk, M.C., Curtis, B., Chrissis, M.B. & Weber, C.V. 1993. The Capability Maturity Model: Version 1.1. IEEE Software, July, 18-27.

Pohl, K., Weidenhaupt, K. Dömges, R., Haumer, P., Jarke, M., & Klamma, R. 2000. PRIME – Toward Process Integrated Modeling Environments. ACM Transactions on Software Engineering and Methodology, 8, 4, 343-410

Rossi, S. & Sillander, T. 1998b. A Practical Approach to Software Process Modelling Language Engineering. In V. Gruhn (Ed.) Proceedings on the 6th European Workshop on Software Process Technology, EWSPT'98, LNCS 1487. Springer-Verlag, 28-42.

Shepard, T., Sibbald, S. & Wortley, C. 1992. A Visual Software Process Language. Communications of the ACM, 35, 4, 37-44.

Sommerville, I., Kotonya, G., Viller, S. & Sawyer, P. 1995. Process Viewpoints. In W. Schäfer (Ed.) Software Process Technology, EWSPT'95, LNCS 913. Berlin: Springer-Verlag, 2-8.

Sorenson, P.G., Tremblay, J-P. & McAllister, A.J. 1988. The Metaview system for many specification environments. IEEE Software, 30, 3, 30-38.

Sutton, S. & Osterweil, L. 1997. The Design of a Next Generation Process Language. In M. Jazayeri, H. Schauer (Eds.) Software Engineering - ESEC-FSE'97, LNCS 1031. Berlin: Springer-Verlag, 142-158.

Sutton, S.M., Tarr, P.L. & Osterweil, L.J. 1995. An Analysis of Process Languages. University of Massachusetts, Department of Computer Science. CMPSCI Technical Report 95-78.

4 CONCEPTUAL FOUNDATIONS OF PROCESS METAMODELLING

Koskinen, M. "Conceptual Foundations of Process Metamodelling".

This paper has been submitted for publication. Copyright may be transferred without further notice and the accepted version may be posted by the publisher.

Conceptual Foundations of Process Metamodelling

Minna Koskinen
University of Jyväskylä

Abstract

Process modelling languages should be carefully selected and adapted to make them suit various needs present in local process contexts. Language standards are mostly inappropriate in such contexts, yet conceptual systematisation is necessary. Process metamodelling is proposed as a means for the specification of process modelling languages in customisable environments. Under a comprehensive process meta-metamodel, different process modelling languages can be consistently specified and process metamodels constructed. The contribution of this study is to develop the conceptual foundations of such an approach. We begin with a discussion of relevant theoretical and linguistic issues, and then continue to developing an integrated model of process metamodels. The proposal is based on a continuum of constructive and experimental work conducted since mid 1990's. Although the work is targeted at metaCASE area, it may benefit anyone who is interested in the conceptual design of process modelling languages.

1 Introduction

Process modelling languages should be carefully selected and adapted to make them suit various needs present in local process contexts: a particular development effort in a particular development organisation. Support for this type of adaptation is necessary in technologies that are customised for a wide range of ISD methods and organisations.

The adaptation of process modelling languages to a specific process context has not attracted much academic interest so far. A more characteristic approach has been a quest for a conceptual standard (Feiler and Humphrey, 1993; Lonchamp, 1993; Conradi et al., 1992; Conradi et al., 1993). Current process support environments offer specific built-in languages with varying designs (Sutton et al., 1995; Conradi and Liu, 1995). Instead, research on evolution and change is targeted almost exclusively at process models (Madhavji, 1992; Bandinelli et al., 1993; Conradi and Jaccheri, 1993; Finkelstein et al., 1994; Kaiser and Ben-Shaul, 1993; Dowson and Fernström, 1994). In metaCASE area, the situation is even worse, since the majority of current metaCASE environments lack process support (Marttiin et al., 1993; Verhoef and ter Hofstede, 1995).

Research on process modelling languages has been technology-intensive, and this may be the greatest reason for the lack of appropriate studies on language adaptation. Yet, there are some case studies that stress the importance of language adaptation in process modelling efforts (Phalp and Shepperd, 1994; Rossi and Sillander, 1998). The first few approaches towards linguistic adaptation of process support (Balzer et al., 1993; Kaiser et al., 1996; Sutton and Osterweil, 1997) are technology-driven and lack adequate theoretical foundations. Sutton and Osterweil (1997) find that a next-generation process modelling language should be able to model processes by composing elements from different language paradigms or representing different semantic aspects.

The justification for language adaptation can be mostly derived from organisational and social considerations, on the role of technology and process improvement in larger contexts. On one hand, it is a question of the impact of a language and its 'universe of discourse' on process thinking. A process modelling language enables the systematic articulation and study of processes, but it also reduces perception of processes and the way in which we comprehend and attack process problems. On the other hand, we are concerned of the impact of process thinking on the personal and collective image of identity, and further on the meaningfulness and sensibility of the social collective within which work is carried out. Incapability to cope with the myriad of ideals and values involved in a given piece of systems development is an inexhaustible source of 'irrational' opposition and conflicts that obstruct process improvement efforts.

Further, it is important to take into account not only the particular forms of process thinking but also their tendency of gradual change. Initiated either by systematic improvement efforts or as an ad hoc response to the necessities at hand, process thinking evolves throughout the development effort in response to individual and organisational learning. As there exists no ideal process thinking that would be appropriate for all organisations, there neither exists one for an organisation throughout all times. Consequently, we are concerned of not only the question of what kind of approach best fits a process context at a given time but also how learning, evolution and improvement are best encouraged and supported.

Due to the general absence of interest in language adaptation, the information systems field lacks a theoretical foundation for language change. Such foundation would enable systematic and rigorous consideration of process modelling languages not only within the technical context but in association with wider social impacts. However, before such a theoretical foundation can be established, we have a more modest goal to achieve: to form a more profound understanding of process modelling languages and mechanisms with which they can be adapted. On one hand, a need to understand the linguistic underpinnings of process modelling has lead us to study the structure of language and techniques. This is necessary to make sufficient abstractions on which further conceptual development can be based. On the other hand, a need to articulate and codify languages and techniques has lead us to study different forms of metamodelling.

Language standards are mostly inappropriate in this context, yet conceptual systematisation is necessary. The need for systematisation should be answered by introducing a generic language framework within which process modelling languages could be designed and constructed in a consistent and integrated manner. We find process metamodelling as a means for context-sensitive specification and evolution of process modelling languages within a given process context. Here, the core of the generic language framework is a generic model of process metamodels.

The idea of process metamodels is not at all new. Already a decade ago researchers argued about the domain (technical vs. organisational), the primary focus (objects, activities vs. decisions) and the dynamics of process models under the title 'process metamodel' (Dowson, 1987). Since then, many attempts have been made to specify a comprehensive set of concepts that could be applied to different development efforts.

Process metamodels have been developed to increase the methodical rigour and applicability of process modelling. Examples of this line of work are NATURE metamodel for requirements engineering (Jarke et al., 1994; Rolland et al., 1995), and Articulator metamodel for software process modelling (Mi and Scacchi, 1990; Mi and Scacchi, 1991) and business process modelling (Mi and Scacchi, 1996). Process metamodels have also been used for environment integration purposes, e.g., to integrate software process and tools (Pohl and Weidenhaupt, 1997), software process and configuration management (Joeris, 1997), a complete specification management environment through processes (Jarke and Rose, 1992), and independently developed tools within generated process-driven environments (Karrer and Scacchi, 1993). In some approaches, process metamodels also codify policies for process model changes (Conradi and Jaccheri, 1993). The strength of explicit process metamodels is their applicability in consistently specifying, composing, and integrating different aspects and parts of a process support environment.

Although process metamodels have been used for establishing better articulated, more rigorous and consistent process approaches, the conduct of process metamodelling itself has not become a well-articulated and rigorous activity. As we will indicate, this is due to two major factors. On one hand, there is a concealed lack of common understanding on metamodelling. On the other hand, current forms of metamodelling are inadequate for properly addressing the specific contingencies that come with process modelling languages.

The core aim of our study is two-fold: first, to enable the contextual specification of process modelling languages and; second, to increase the rigour, contextual sensitivity, and comparability of process modelling approaches by increasing the rigour of process metamodelling. To our knowledge, comparable work is not carried out elsewhere. The objective of this work is to establish a conceptual foundation for process metamodelling. We begin with theoretical and linguistic issues. First, we discuss the structure of language and techniques (Section 2), and study different metamodelling approaches to clarify the perspective we adopt (Section 3). Thereafter, we continue to develop an integrated conceptual model of process metamodels

(Section 4) and consider some issues related to modelling techniques (Section 5). Finally, we draw some preliminary conclusions (Section 6).

2 Language and Techniques

A *language* is a systematic means of communicating ideas or feelings by means of conventionalised signs, sounds, gestures, or marks that have understood meanings. Language is, for us, a basis for communication on matters as much as for comprehending them. As a means of *communication*, language is a heritage of a collective. It is a convention for sharing information. The arena on which language is shared and endured, is an arena of public. As such, language confines to *expression* (representing something in a medium) and *interpretation* (grasping what someone intends with an expression). As a means of *comprehension*, again, language is a private enterprise. It is learnt, adopted and assimilated into an individual mind. Language is a device of (re)constructing concepts abstractly and explicitly, thereby giving an individual a way to grasp the world. As such, it contributes to *perception* (recognising and noting facts and occurrences), and *conception* (the mental capacity of forming and understanding ideas, abstractions, and their symbols).

Our focus of interest, of course, is not on natural language but a certain form of artificially constructed language. Yet, a process modelling language has no relevancy out of the context of natural language. A process modelling language is – or should be – as much influenced by natural language as it itself influences natural language. It is not a mere artificial aid but is interwoven into one's everyday jargon and local linguistic heritage. This is in contrast to process programming languages that are intended only for instructing a mechanical interpreter.

In our quest for a useful conception of language, we first have to look for a suitable linguistic model, on which we can base further conceptual work. We choose to start with Ullman's triangle (see, e.g., Baldinger, 1980). This 'semiotic triangle' has been applied in several earlier studies in IS field (e.g., Bergheim et al., 1989). This model comprises three interrelated entities: *names* that symbolise *concepts* that refer to *things* in the world. Even though Ullman's triangle was not originally intended as such, it can be conceived of as "a methodological model on the level of the second metalanguage (level of linguistic methodology)" (Baldinger, 1980). As such, it suits our goal to outline a unified view of the world of concepts, signs and meanings[1].

[1] A discussion of the evolution of the 'semiotic triangle' can be found, e.g., in (Baldinger, 1980). Within general linguistics, also more elaborate models (e.g., the trapezoidal model) have been developed but the enhancements are made mostly with regard to speech. (See an account for semantic theories in (Baldinger, 1980)). In the present scope we find them unnecessarily complex. The 'semiotic tetrahedron' (Falkenberg et al., 1998) is developed for the specific needs of IS field on the basis of the 'semiotic triangle'. It distinguishes between a *domain* (i.e., referent), a *conception*, a *representation*, and an *actor* (or observer in (Braun et al., 1999)). As such, the model contributes not so much to a structured conception of language, but to a pragmatic conception of how we comprehend and use signs.

There is a considerable diversity of opinion among linguists about what the terms 'sign', 'concept' and 'meaning' precisely denote and how they relate to each other. Especially, the very nature of 'meaning' is elusive and the principal question examined by semanticists has been that of how we are to conceptualise meanings. Not having generally agreed denotations in linguistics, we describe the terms here for our limited purposes. The following characterisations may thus be theoretically disputable in linguistics but hopefully yet illuminating and useful for IS researchers.

(1) A *concept* is something conceived in the mind. It is formed through discrimination, by finding there "something else" in the world that one can not comprehend as something he or she already knows and understands. Thus, a concept is constructed in comparison to other concepts: by determining how it is distinguished from them. The concept is then made expressible by giving it a *name* and showing how the name is used.

(2) A *sign* is a fundamental linguistic unit that symbolises a concept or has a purely syntactic function. As a designator, it is the *name* of a concept.

(3) A *meaning* of a concept is the result of the concept being actualised in the world. Thereby this result – a "*thing*", widely understood as any phenomenon or complex of phenomena – becomes the *referent* of all signs that are perceived to stand for that particular "thing".

Respectively, a language consists of three intertwined systems: a *conceptual framework* that is a system of concepts, a *notation* that is a system of signs, and a *semantics* that is a system of meanings. In the following, we discuss the structure of process modelling languages based on this view (Section 2.1). Thereafter, we extend our discussion to techniques as composites of a language and an operational semantics (Section 2.2). Furthermore, we show a contrast to process programming languages.

2.1 The Structure of Process Modelling Languages

A process modelling language composes a conceptual framework for the composition, a notation for the representation, and a semantics for the interpretation of process models (cf. Lonchamp, 1993). It is arguable whether any distinction can be drawn between a system of concepts and a system of meanings (Fodor, 1977; Baldinger, 1980). However, process modelling languages are unique in the sense that they are intended for dual interpretation: human and automated. Therefore, it should count for a conceptual framework for human comprehension and a semantics (at least) for automated enactment.

Conceptual framework. A conceptual framework is a mental structure that gives us a means to focus, structure and organise our perceptions of the world. Thereby it determines which phenomena and signs are meaningful to us. Since our conceptions form a basis for our comprehension and thereby our knowledge of the world, a conceptual framework is also a major force affecting the formation of our knowledge. A conceptual framework establishes a

conceptual system that consists of a set of concepts and their interdependencies. This system underlies the representation, interpretation and reasoning of any model. The conceptual framework of a process modelling language determines in which terms and ways we abstract, discuss and reason about processes.

Notation. A notation is a system of signs or symbols related to a conceptual framework, and it contains rules for the arrangement of the signs into meaningful wholes (e.g., sentences). A process modelling notation follows a representation style that determines what kinds of notational construct are available and how they can be related to each other. Examples of representation styles are structured text, diagrams, forms, calendars, matrices, tables and hypertext. Examples of notational constructs are graphical characters and symbols, fields, rows, columns, and links[2]. A specific notation forms a style-specific representation scheme according to which processes are represented. A representation scheme determines how style-dependent notational constructs are explicitly represented and how they relate to the conceptual framework. Examples of these issues include: on what grounds different concepts can be represented as notational constructs; how the concepts are represented; and which notational rules apply to such representations.

Semantics. Semantics concerns the meaning of signs. In AI research, the term 'semantics' denotes "some form of correspondence specified between a surrogate and its intended referent in the world" (Davis et al., 1993). As we understand semantics as a form of subjective knowledge, we refine (and redefine) this view. The correspondence between a surrogate and its intended or actualised referent in the world is established through subjective knowledge of meaning. Even in case of a machine interpreter, the semantics of a language are rules codified in the interpretation mechanisms of the interpreter. The interpreter is the agent that actualises meanings for representations. Thus, the term 'semantics' denotes either human knowledge (in human oriented interpretation) or computational rules (in machine oriented interpretation), according to which the meaning of a representation is established.

This three-fold view of language should be familiar to those who study language design. Yet, its implications on the structure of languages are not adequately accounted for in current research. A process modelling language can be viewed as a complex system of interdependent language constructs. Firstly, conceptual constructs consist of those that specify the conceptual framework of a language. They give the language a specific 'ontology'. Secondly, notational constructs consist of those that specify the notation of a language. They give the language a capability to represent. Thirdly, semantic constructs consist of those that specify the semantics of a language. They make the language enactable. Through these three kinds of constructs, a process modelling language becomes a means to compose, represent, and enact process models.

Current theoretical developments on language constructs lack this structure. Hence, the view of process modelling languages taken in individual

[2] Note that representation styles and notational constructs are understood here abstractly, not as implementations. The same style or construct may be implemented in different ways.

studies may vary greatly. The practical outcomes of different classifications are not easy to compare, since the most fundamental distinctions they make between language constructs are often contradictory. In the data modelling area, a similar distinction between representation and conceptual type-level constructs has been made (Tolvanen & Lyytinen, 1993), but without considering the further practical refinements and implications of the notion.

2.2 The Structure of Modelling Techniques

A modelling technique complements a modelling language with a mechanism that produces and manipulates the models. The rules of this mechanism are known as an *operational semantics*.

This should not be confused with an operational semantics of a programming language (see, e.g., Meyer, 1990), although they share a common origin. The idea of operational semantics originates from a linguistic school called operationalism (see, e.g., Hardy, 1978). To reduce the ambiguity of definitions, operationalism describes them in terms of operations that can be unequivocally performed. The *operational meaning* of a concept is the set of operations that are performed to bring about a phenomenon. If the operations (actual or possible) vary, also the meaning varies. Congruently, an operational semantics defines a system of operations according to which a certain kind of phenomenon is produced. Although one rarely encounters the term 'operational meaning' in the current literature, the term 'operational semantics' has firmly rooted its usage in computer science – specifically in relation to programming languages.

The operational semantics of a technique consists of a set of rules for model creation and evolution. In a sense, it adds a modelling technique with its necessary "process view" by determining operations applicable on a model (e.g., add, modify, delete). Besides mere model construction, an operational semantics may also concern such issues as modelling rationale, guidance and traces, component reuse, model configuration and versioning, and modelling transactions. That is, the operational semantics of a modelling technique covers everything that intimately relates to modelling with the technique and should be performed when the technique is used.

A major aim of such a comprehensive operational semantics is to maintain consistency during model creation and evolution. Thereby it is a prominent vehicle of comprehensive change management, independent of the origin or purpose of change. In this respect, it is of utmost importance that change management is not restricted to configurations, versioning and transactions. The above areas are, however, usually managed separately in different domains. An operational semantics should instead integrate all the areas from rationale to transactions so that any operation that is carried out has a consistent outcome.

2.3 A Contrast to Process Programming Languages

It useful to clarify the contrast between process programming languages and enactable process modelling languages. A programming language is a notation for giving instructions for an execution mechanism. A programming language is understood to comprise syntax and semantics, where syntax forms the foundation on which various semantic constructions are built. It does not distinguish conceptual frameworks (although there could be certain advantages of doing this).

There are different approaches to semantics that complement each other in the process of specifying and implementing a programming language[3]. The one we are specifically interested in is operational semantics. The idea of operational semantics is to express the semantics of a language by giving a mechanism that makes it possible to determine the effect of any program in the language. An operational semantics is a set of transition rules specifying how the state of this mechanical interpreter changes while executing a program. The operational meaning of a program is the consequent sequence of interpreter states.

A process engine is a mechanical interpreter that uses a process model or process program as its input. Usually, though, process models are first translated into a process program to make them enactable. Language semantics[4] is interwoven into the operational semantics that is encoded into a process engine. There are currently only few approaches for configuring a process engine to change the way of enactment without making changes to process models (Balzer & Narayanaswamy; Kaiser et al., 1996). However, these approaches are based on extending an existing language, not embedding a new one.

There are two consequences of customisable process metamodels on a generic process engine. First, the language semantics needs to be specified apart from a process engine. Second, the process engine needs to be capable of using an explicit process metamodel as its input data. It would interpret process models while using an explicit language specification to guide this interpretation. By distinguishing a conceptual framework from notation and semantics, we achieve an explicit, integrating medium between the latter two. Furthermore, we can adopt the conceptual framework as the foundation of language construction[5], which results both in notational and semantic flexibility.

Such a generic process engine needs to distinguish between a language semantics, an enactment mechanism, and a reflection mechanism, each of which determines only a part of its functionality. A language semantics provides rules according to which a process engine interprets process models, but it does not

---

[3] An introduction to the theory of programming languages can be found in (Meyer, 1990).

[4] 'Language semantics' refers here to semantics as discussed in Section 4.2.1. We use the term in this section in order to avoid terminological confusion.

[5] Consequently, it is not a representation that has a meaning, but a concept that has both a representation and a meaning. In this way, a concept may also have multiple representations and multiple meanings within explicitly specified (and formal) constraints.

specify how the interpretation mechanism works. An enactment mechanism determines how the matching of the language semantics with a particular process model is carried out and how it is enacted. The language semantics is separately embedded in the enactment mechanism. The mechanism implements an enactment pattern, i.e., a set of generic enactment operations capable of handling the combined data from a process model and a process metamodel. Furthermore, the functionality of a process engine encompasses more than interpretation and enactment. A reflection mechanism determines how a process engine changes or allows the change of a process model according to the operational semantics of a modelling technique (discussed earlier in Section 2.2).

3 Metamodelling Approaches

Metamodelling is a form of specification, in which information of a class of models is collected and codified into a metamodel. Since metamodelling is a domain independent mechanism, it can be applied in a multiple ways. Indeed, that is what has happened in the past. The usefulness of metamodelling is widely recognised (Mili et al., 1995; Jarke et al., 1998). It is extensively used in data modelling frameworks, e.g., for a higher degree of data integration in CASE environment frameworks (Chen and Norman, 1992) and for the specification and customisation of metadata in metaCASE environments (Marttiin et al., 1993; Marttiin et al., 1995; Verhoef and ter Hofstedte, 1995). Currently, the interest in metamodelling is increasing both in academia and industry.

Process metamodelling can be used as a means for language adaptation, by which the process metamodel underlying a process modelling and support architecture can be changed. Well-articulated, rigorous and consistent process approaches can be established with explicit process metamodels. Process metamodelling itself, however, lacks a foundation and has not yet evolved into the state of a well-articulated and rigorous activity. We find reasons for this to be twofold. On one hand, the conception of metamodelling varies largely between different research efforts but this variation is difficult to notice without a detailed study. Therefore it is difficult to compare differences and to create common understanding necessary for more comprehensive developments. On the other hand, process metamodelling has not established its place as an independent form of metamodelling. Yet, the domain-specific contingencies that arise with process modelling languages are not accounted for by 'ordinary' forms of metamodelling and hence not properly addressed.

There is a general agreement among researchers on what metamodelling is. When examined more closely, however, the term appears to have many uses. A representative set of examples can be found in the workshop summary of Metamodelling in OO (Mili et al., 1995). Sometimes the difference between two uses can be detected only in an in-depth study and comparison. Two

research groups may promote different notions of metamodelling and yet debate without noticing the fact. Therefore, we find it compulsory to examine and clarify different senses of metamodelling and locate ourselves in this maze.

In the following we take two aspects of metamodelling into scrutiny: the base domain of modelling and the modelling dimension.

3.1 Base Domains of Modelling

Every form of modelling has a specific target domain that we call a *modelling domain*. A modelling domain can be, for example, information systems for systems modelling, a system model for (systems) metamodelling, a method for method modelling, or a systems development process for process modelling. A *base domain of modelling* is any modelling domain the purpose of which is not to represent information of models. For example, information systems form a basis for systems modelling and metamodelling, and development processes for process modelling and process metamodelling. A base domain is thus the root of a modelling level hierarchy (i.e., model, metamodel, meta-metamodel, etc.).

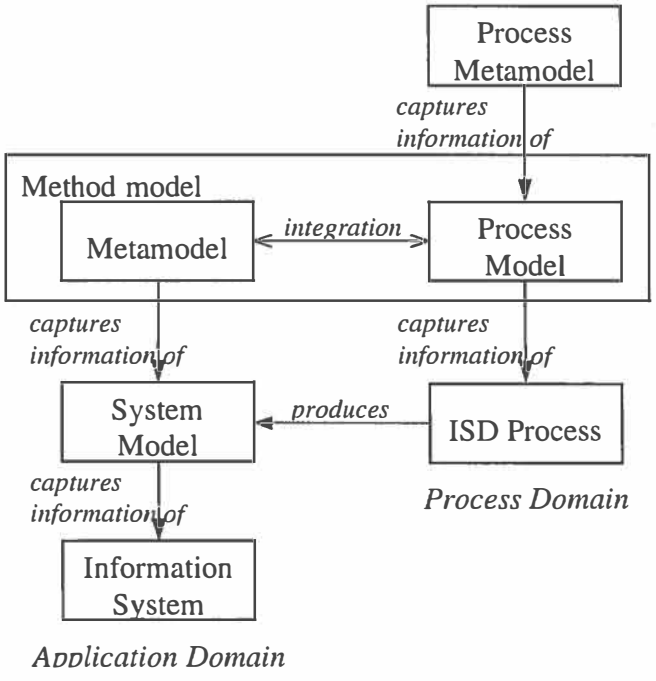

FIGURE 1 There are two interrelated base domains related to systems development: application domain and process domain.

The base domain of IS modelling and (ISD) method engineering is information systems ('application domain'). Method engineering develops techniques for systems modelling and uses metamodelling for their specification. Similarly,

the base domain of (ISD) process engineering and PML engineering[6] is the systems development process ('process domain'). The relationship between the two domains is shown in figure 1. Models of an information system in the application domain are produced through a development process in the process domain. Metamodels (i.e., models of system models) and process models (i.e., models of development process) are specified and integrated into method models. Process metamodels (i.e., models of process models) are specified as part of meta-methods.

3.2 Modelling Dimensions

Most metamodelling approaches to date have their origin in the application domain but differ in regard to the dimensions they consider. The modelling dimension determines what kind of information metamodelling is supposed to capture. There are major differences between metamodelling approaches in this regard. These differences mostly reflect the emphasis of a given subject of study, e.g., concepts in data base and specification management research, notations in the research on model representation, or techniques in tool construction.

Broadly speaking, any kind of model related information can be chosen as a modelling dimension. The only constraint is that this information is created through the abstraction of features common to a class of models. That is, the characteristics should replicate across all possible models of that kind, independent of what the model represents. In other words, metamodelling codifies some type of information incorporated in modelling languages and techniques. In the following we refer to this information as 'meta-information'.

In accordance to our earlier discussion of language and techniques, we distinguish four parts that can serve as a modelling dimension individually or in combination: notation, ontology (i.e., conceptual framework), semantics, and operational semantics. The common approaches to date cover notations and conceptual frameworks.

The major metamodelling approaches can be currently classified according to their dimension as ontology-based, language-based, or technique-based. First, an *ontology-based* metamodelling approach is used for multi-level ontological abstraction of real-world phenomena (see, e.g., Nissen et al., 1996). Here, metamodelling is seen as the multiple instantiation levels of application knowledge with a focus on conceptual frameworks. Second, *language-based* metamodelling approach is used for the specification of modelling languages. Such approaches most often codify a notation together with some degree of conceptual meta-information (see, e.g., Kelly et al., 1996). In contrast to ontology-based metamodelling, ontological abstraction is represented as ancestor type hierarchies (subtype – supertype) instead of metatype hierarchies (type – metatype). Third, a *technique-based* metamodelling approach is used for the specification of modelling techniques, where operational semantics is

[6] PML engineering concentrates on the design of languages and techniques for process modelling (Rossi and Sillander, 1998).

considered together with the modelling language (see, e.g., Sorenson et al., 1988). Metamodelling is thus not limited to the information of models but it also codifies information of the process through which a model can be constructed. However, it differs from fine-grained process modelling in that the codification abides more to a declarative than a procedural format.

The difference between ontology-based and language or technique-based approaches is important. Both use similar vocabulary at the lowest metalevels, but with different denotations. The use of metalevels is also most often restricted to the first, fixed metamodel, or second, fixed meta-metamodel. An approach is easily misclassified unless it is carefully analysed. Common with the approaches is that a *metamodelling language* is used to specify any M^n-model[7]. However, in ontology-based metamodelling any M^n-model defines *what information systems are*, and a metamodelling language defines *how information system ontologies are represented*. In language or technique-based metamodelling a metamodel defines *what system models consist of*, and a metamodelling language defines *how languages or techniques are represented*.

The highest metalevel model is implemented in a support environment. In ontology-based approaches, this model is known as an 'omega-level' model (Nissen et al., 1996), whereas in language and technique based approaches it is known as a meta-metamodel. On the other hand, in ontology-based approaches a 'meta-metamodel' denotes an M2-level model. Thus, it is the 'omega level' and not the 'meta-metalevel' of an ontology-based approach that is comparable to the 'meta-metalevel' of a language or technique-based approach.

In the following, we focus on language-based process metamodelling while still keeping the approach open to later extension into a technique-based one. It is important to note, however, that ontology-based approaches introduce more efficient mechanisms to represent domain ontologies. An ontology-based omega-level model and the related metamodelling mechanisms can provide substantially more support for ontological constraints (see, e.g., Jarke et al., 1998) than a generic language or technique-based approach. To be as powerful, the latter should incorporate similar mechanisms for the specification of ontological type hierarchies. Enhancing current language and technique-based approaches with such mechanisms forms a relevant and interesting research topic.

4 A Conceptual Model of Process Metamodels

Process metamodelling collects meta-information of process models and represents it in a process metamodel. That is, process metamodelling captures information incorporated in a process modelling language or a technique. A *process meta-metamodel* is a model that captures meta-information of process

[7] Figure n denotes the order of metalevel. For example, M1 model is a metamodel and M2 model is a meta-metamodel (see Nissen et al., 1996).

metamodels and, thereby represents information incorporated in a process metamodelling language.

The first step in designing support for process metamodelling is to develop a comprehensive set of metaconcepts with which different process modelling languages can be designed. Thereafter, metaconcepts need to be integrated with an explicit notation and semantics to construct a process metamodelling language. This language is then specified as a process meta-metamodel and implemented into a process modelling environment. The goal of the present work is to specify a set of metaconcepts needed to specify and integrate different combinations of conceptual frameworks, notations and semantics. We want to avoid an excessively generic approach that would make process metamodels too complex and difficult to maintain and evolve. One main objective is to make process metamodelling as easy as possible without compromising its expressive power.

We specify the structure of process metamodels according to our discussion of the structure of language and techniques. First, we redefine the term 'process metamodel' from its ontology-based counterpart (Lonchamp, 1993). A *process metamodel* is a model of meta-information about a class of process models. To follow a language-based approach, we have to be able to represent both conceptual frameworks, notations and semantics. For this purpose, a *language-based process metamodel* integrates three sub-parts. A *conceptual process metamodel* represents the conceptual framework, a *notational process metamodel* the notation, and a *semantic process metamodel* the semantics of a process modelling language.

In the following sections we develop an integrated conceptual model of process metamodels, which includes a model of conceptual process metamodels (Section 4.1), a model of notational process metamodels (Section 4.2), and a model of semantic process metamodels (Section 4.3).

4.1 A Model of Conceptual Process Metamodels

Although, in minimalist terms, the term 'process' simply denotes the progress of something, it cannot be understood without saying something about the context within which the progress is brought about. A conception of a process is always arbitrarily formed. Progress, and the context within which the progress occurs are determined by the conceptual framework underlying the observers' thinking. A conception of a process includes those aspects that people regard meaningful for understanding, analysing and conducting progress.

A process interacts with its context and makes changes to the context in order to reach its goals. Again, what constitutes a process and what its context is arbitrary and depends on one's conception. Feiler and Humphrey (1993) define a process as "a set of partially ordered steps intended to reach a goal". As well, we could say that a process is "a set of interdependently emerging situations to be acted upon by an agent to meet the requirements set upon them". Although the latter phrase evokes a different conception from the one

evoked by the former, they indisputably refer to the same real-life phenomenon. The context of a process may be comprehended as any set of organisational, linguistic and technological structures, and constraints such as goals, policies, languages, data, roles and resources (cf. Lonchamp, 1993). On the other hand, any of those may as well be understood as part of the process.

Either way, a process model that codifies this conception is only a partial representation of the total complexity. It has a focus only on limited aspects of the whole and as such its effectiveness depends on the practical situation within which it is used. To be able to account for a wide range of conceptual frameworks, we need to distinguish a set of generic concept categories and their interdependencies. The following discussion intends to aid in this.

Process elements are entities that refer to atomic actions, or compose a set of other process elements (Feiler and Humphrey, 1993). The substructure of atomic actions is not made explicit in process models. Process elements have interdependencies, most of which relate to co-ordination and progress. It is often difficult to determine a clear boundary for a composite process element, since any composition is arbitrary in relation to practice. Therefore, we can not expect a decomposition hierarchy necessarily to form a clearly bounded whole. Process components (clearly there are more to a process than process elements) share a context from which they obtain properties. These properties may be immediately attributed (such as 'name' to 'task'), but also appear as a reference to a conceptual entity (such as 'data file').

The above concerns a conception of human processes ('user process'). Within the context of automated enactment, we also have to construct a conception of the automated process ('environment process') that supports the human process. Such a process is a series of automated operations – a true "partially ordered set of steps" – that are ensued (directly or indirectly) from human actions, e.g., taking menu options. This process comprises all actions that manipulate electronic data: calculate values, propagate constraints, and so forth. Another concept, one that is necessary for keeping track on progress, is the state model. It comprises a conception of the life-cycle of individual process elements. A conception of an environment process is only necessary to the degree it is used for human comprehension. Otherwise, it is sufficient to address it in language semantics.

In the following we discuss the classification of conceptual constructs and dependencies incorporated into a conceptual model of conceptual metamodels. The model is shown in figure 6. There are two things that must be noted. First, it is somewhat misleading to label the categories with meaningful names since it makes one think that the semantics of a category or what it collects is somehow bound with the meaning of its name. The model could be easily mistaken as a generic process metamodel. However, conceptual constructs are not concepts but constructs representing concepts.

The difference between a process metamodel and a process meta-metamodel is the way how their categories are constructed. A process metamodel is based on factual categories that are established according to specific characteristics of the items categorised. Due to their empirical nature, they are more or less contingent and incomplete. In contrast, a process meta-

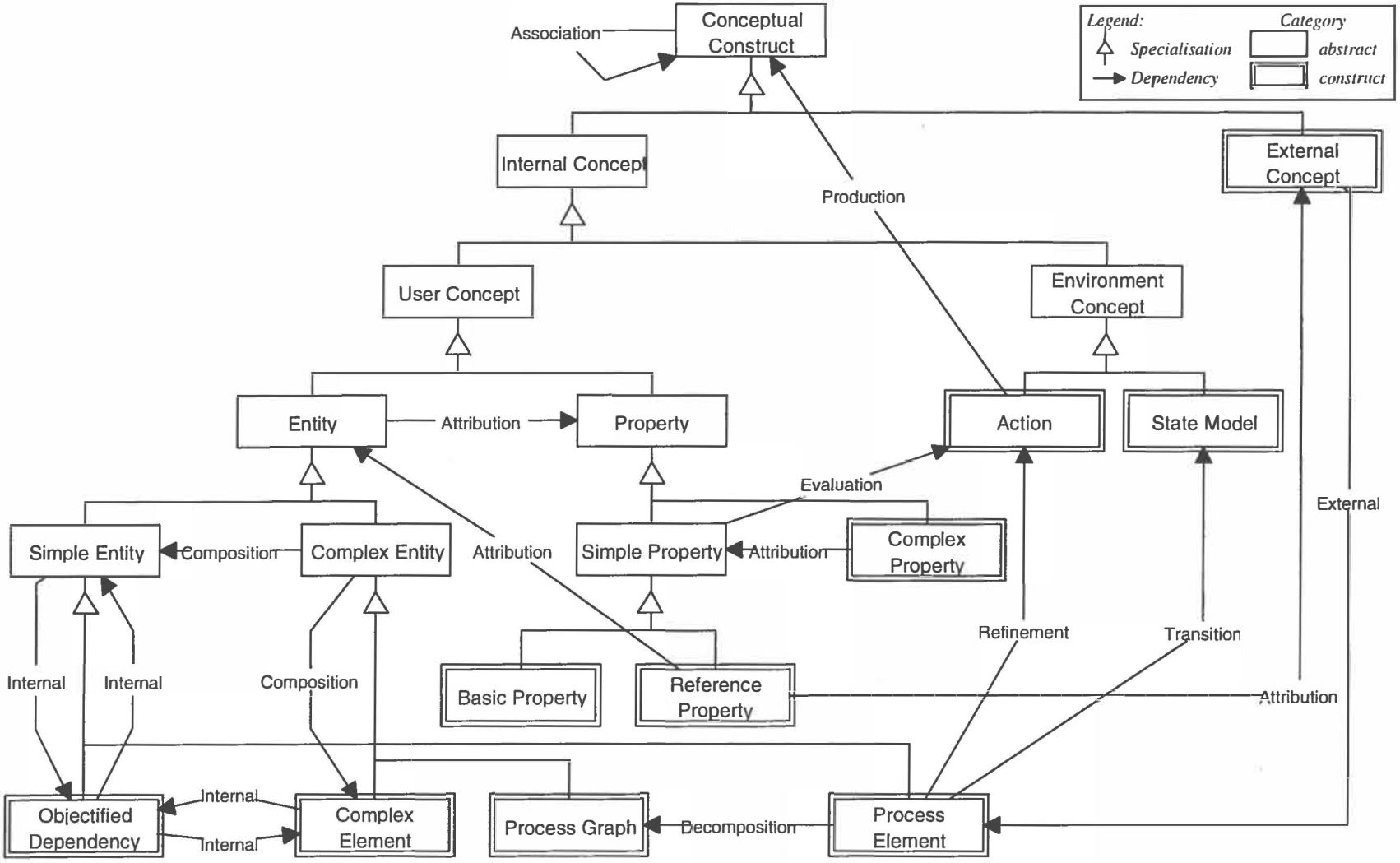

FIGURE 6 A conceptual model of conceptual process metamodels.

metamodel has to conform to logical categories that are derived by a systematic use of dichotomy. In dichotomy one abstract category is divided into two exhaustive and exclusionary categories. Creation of a category tree begins with one all-encompassing root category, and ends with a set of finer leaf categories. Consequently, it applies that every concept that can be placed in the root category, can be placed in one and only one leaf category. To be all-encompassing, it has also to apply that no concept can be placed outside the root category. This ensures that the Universe of Discourses (everything that ever can be stated) can be covered by the leaf categories.

4.1.1 Conceptual Construct Categories

We distinguish two generic categories of conceptual constructs, which discriminate between a process and its context. First, a concept of process is constructed of *internal concepts* that form the core of a process model. These are further divided into two categories that discriminate between a user process and an environment process: *user concepts* and *environment concepts*. The former category distinguishes between *entities* that are conceptually autonomous (such as 'task'), and *properties* that are attributed to entities (such as 'duration'). Second, the concept of the context of process (to the degree it is necessary to interact with it) is constructed of external concepts. Note that external concepts may designate both technical and non-technical "things".

Entities are divided into two categories regarding whether they are atomically conceived, or not: *simple entities* and *complex entities*. Note that being "atomically conceived" does not imply that an entity really is atomic. For example, a 'procedure' may be atomically conceived, yet non-atomic in the sense that it is associated to a non-atomic concept of its decomposition, a 'sequence of steps'.

Like simple entities, we discriminate between the substance of a process component and a dependency that can be conceived as an entity on its own between such substances: *process elements* and *objectified dependencies*. We distinguish between two categories of complex entities regarding whether an entity is autonomously applicable as a concept of a process, or dependent on being embedded into a more general concept: process graphs and complex elements. *Process graphs* are entities with clearly conceivable boundaries. Whatever dependencies a process graph has, they all are established at the level of the whole (cf. "interface"). Thus, it is applicable as a concept of a process as a whole. *Complex elements*, on the other hand, are concepts with less clear boundaries. A complex element has dependencies both at the level of the whole (which justifies it being a concept on its own) and at the level of its parts (which makes its boundaries somewhat artificial).

Also properties are divided into two categories based on whether they are atomically conceived or not: *simple properties* and *complex properties*. Simple properties are further discriminated on the basis whether they qualify as immediate or referencing properties: *basic properties* and *reference properties*.

Although being atomically conceived, a reference property may associate to a non-atomic concept. *Complex properties* are concepts of property composition.

Since environment concepts are needed only to the degree users need information of the environment process, we distinguish only two such categories. First, an action is a concept of an automated operation. Second, a state model is a concept of the enactment life-cycle of a process element. We do not consider here the components of state models (although it is obvious that they are composites), since there are different, equally applicable ways to distinguish them. The role of a state model, however, is to associate different types of event to the different phases of an enactment life-cycle.

4.1.2 Conceptual Dependency Categories

Conceptual dependencies can be classified according to the scheme shown in table 1. First, we distinguish between formal and informal dependencies. Formal dependencies have specific semantics, whereas informal dependencies are based on a free association. Secondly, formal dependencies are divided into *structural dependencies* that appear between user concepts, and *executive dependencies* that appear between user concepts and environment or external concepts. The latter form an executive link between user processes and environment processes.

TABLE 1 A classification of conceptual dependencies.

	Formal		Informal
	Structural	Executive	
Lateral	internal	external	association
Hierarchical	composition decomposition	refinement	
Contextual	attribution	evaluation production state	

On the other hand, conceptual dependencies can be classified in regard to their structural loci. Lateral dependencies appear on the plane of a unified abstraction level, on which we can establish such concepts as 'sequence' and 'concurrency'. Hierarchical dependencies appear in terms of reducing or increasing abstraction between lateral planes. Contextual dependencies appear when contrasting and specialising a concept against its context by a specific information content.

We distinguish among four categories of structural dependencies. First, *internal dependencies* build up the lateral structure. They relate objectified dependencies to process elements or to other objectified dependencies. Secondly, the hierarchical structure is created through *decomposition dependencies* that integrate process elements with a finer-grained decomposition structure in

process graphs, and of *composition dependencies* that produce the composite structure of process graphs and complex elements. Both process graphs and complex elements are composed of process elements, objectified dependencies, and other complex elements. Thirdly, *attribution dependencies* build up the contextual structure by associating a set of properties to an entity or a complex property. A reference property can be further associated to an entity or an external concept.

We also distinguish five categories of executive dependencies. Firstly, the lateral structure is augmented through *external dependencies* that create a connection between a process and its context. They locate between process elements and external concepts. Secondly, the hierarchical structure is supplemented through *refinement dependencies* that create a supporting link between a user process and an environment process. They appear between process elements and actions. Thirdly, the contextual structure is amplified through three types of dependency. *Evaluation dependencies* relate a property with a calculation or an inference. They appear between properties and actions. *Production dependencies* create a link between an operation and a consumed or produced object. They appear between actions and any kind of conceptual construct. Note that when a reference property is referenced as an object, it is indeed the property not its referenced value that is referenced. That is, the link is reflective. *State dependencies* associate a process element with its life-cycle state. They appear between process elements and state models.

Lastly, we distinguish only one category of informal dependencies, since there is no generic difference between lateral, hierarchical, ad contextual associations. Association dependencies create informal and possibly ad hoc conceptual relationships between arbitrary concepts, e.g., to add links, comments, questions, or rationale.

4.2 A Model of Notational Process Metamodels

Notational constructs differ from conceptual ones in that the constructs can be classified from two points of view. First, they can be classified according to their role in a representation system. Second, they can be classified according to their representation style(s). For conceptual constructs, there are no "conceptual styles" to consider.

The systemic classification yields a set of generic notational constructs for process modelling languages, which are replicated across representation styles. Traditional representation styles include structured text and diagrams, but there is not reason to exclude such common styles as forms, calendars, matrices, tables and hypertext, either. The emergence of Internet and multimedia further enriches the set of potential alternatives. A representation style instantiates the generic notational constructs and dependencies, thereby determining what kinds of notational constructs are available and how they can, or must, be related in that style.

Some shortcomings in the traditional thinking about representation styles need to be pointed out. First, representation styles are often misconceived to be clear-bound; that diagram representations are composed of symbols and lines,

and textual representations of text. Today, different representation styles share more and more notational constructs, and are increasingly integrated as complementary forms of representation. Hypertext and multimedia are fine examples of this tendency. Secondly, it is often assumed that one could find and enumerate a finite set of notational constructs to use within a representation style. Yet, new means of representing emerge, especially along technological advances. It would certainly be an advantage to be able to deploy them also in process modelling. Thirdly, it is also assumed that notations account only for a part of a visualisation system. For example, it seems obvious that there is some notation underlying a diagram representation of a process, but not that there is one also for a form, or a dialog that prompts for some specific information of the process, or for a hyperlink between a diagram and a form. Instead, all parts of a visualisation system are necessarily accounted for by a notation, if they are to make sense at all.

Due to this inevitable lack of clear bounds and stability, it is not possible to outline an exhaustive set of notational construct categories. New notational construct categories also require new mechanisms for the generation of tool support. If a metamodelling system is based on a set of notational construct categories, the style gallery would be difficult to extend safely. Hence, we find it better to determine a set of generic categories of notational constructs, based on representation styles that can be specified and extended. This way, extensions will not jeopardise the consistency and integrity of the metamodelling system.

4.2.1 A Generic Model of Representation Styles

In figure 2, a model of generic notational construct categories is shown. Firstly, *views* are constructs that compose an independent representation that is usually shown separately, whereas *visual fragments* are composed into a view and shown together. Secondly, visual fragments are distinguished into view fragments, interface fragments and visual dependencies. *View fragments* are visual entities fully determined within the notational system. In contrast, *interface fragments* are visual entities that connect the representation system to an external system, and hence their structure is externally determined. *Visual dependencies* are fragments that combine visual fragments. Thirdly, *visual attributes* are different qualities of views and view fragments. A visual attribute does not extend a representation, but gives it a specific form of appearance.

There are also five categories of generic notational dependencies shown in figure 2. Firstly, *inclusion dependencies* make visual fragments appear within a larger context, i.e., a view, or a visual dependency. The appearance of an included fragment is dependent on the appearance of its context. The context is not similarly affected by changes of its included fragments. For example, a change in the position of a view would entail a similar change in the position of included fragments, but not vice versa. Secondly, *attachment dependencies* 'glue' visual constructs together. In contrast to inclusion, the dependency is mutual. The position of attached fragments could not be changed independently of the other fragment nor vice versa. Thirdly, *connection dependencies* make connected fragments affect the appearance of each other. For example, a change in the

position of one connected fragment might change the appearance of the other. Fourthly, *characteristic dependencies* relate views and visual entities with visual attributes to give them a special outlook, such as colour. Fifthly, *linkage dependencies* appear as guided changes of focus between visual entities. We give some examples of specialised categories in Section 4.2.3.

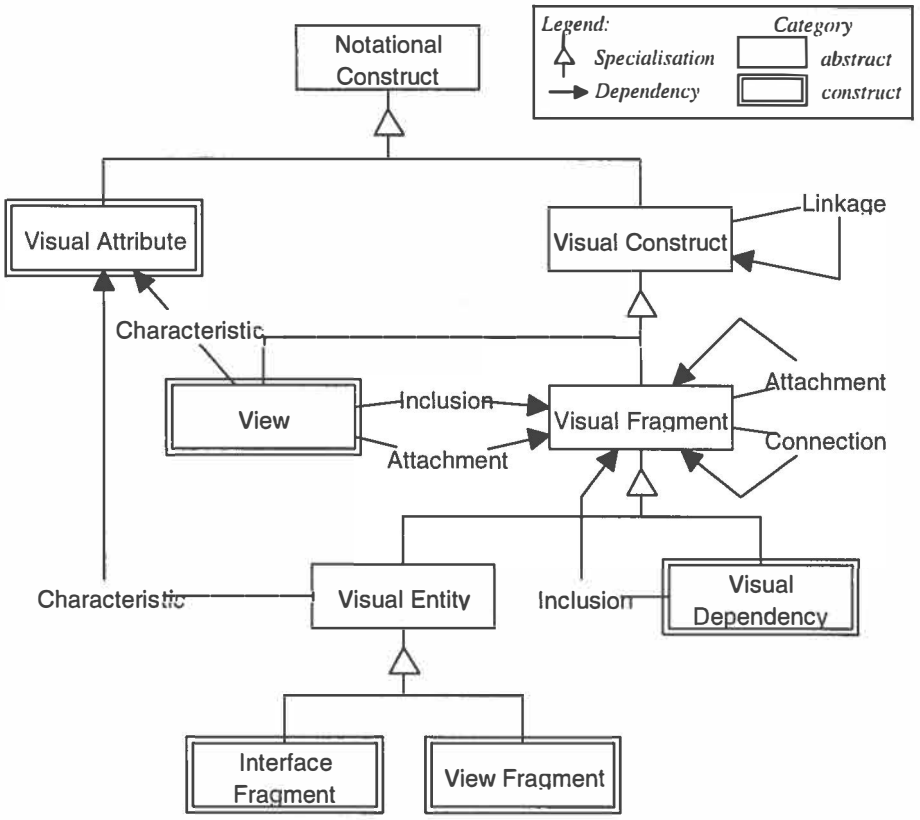

FIGURE 2 A generic model of representation styles.

A representation style allows a variety of potential notations. A notation relates the constructs of a representation style with a conceptual framework. All conceptual categories are usually not visualised within a single notation. For example, many diagram representations concentrate on a subset of process elements, objectified dependencies and decomposition, whereas table representations might focus on process elements and their properties. A notation specifies how process models are visualised (e.g., the actual symbols) and which notational rules apply to them (e.g., how different symbols may appear in relation to each other, and within which conditions they may appear).

Representation styles are formed by applying the generic categories in a representation context, such as diagrams or forms. An example of a combined diagram and form-based representation style is given in figure 3. In the following sections, we discuss the categories of notational constructs and

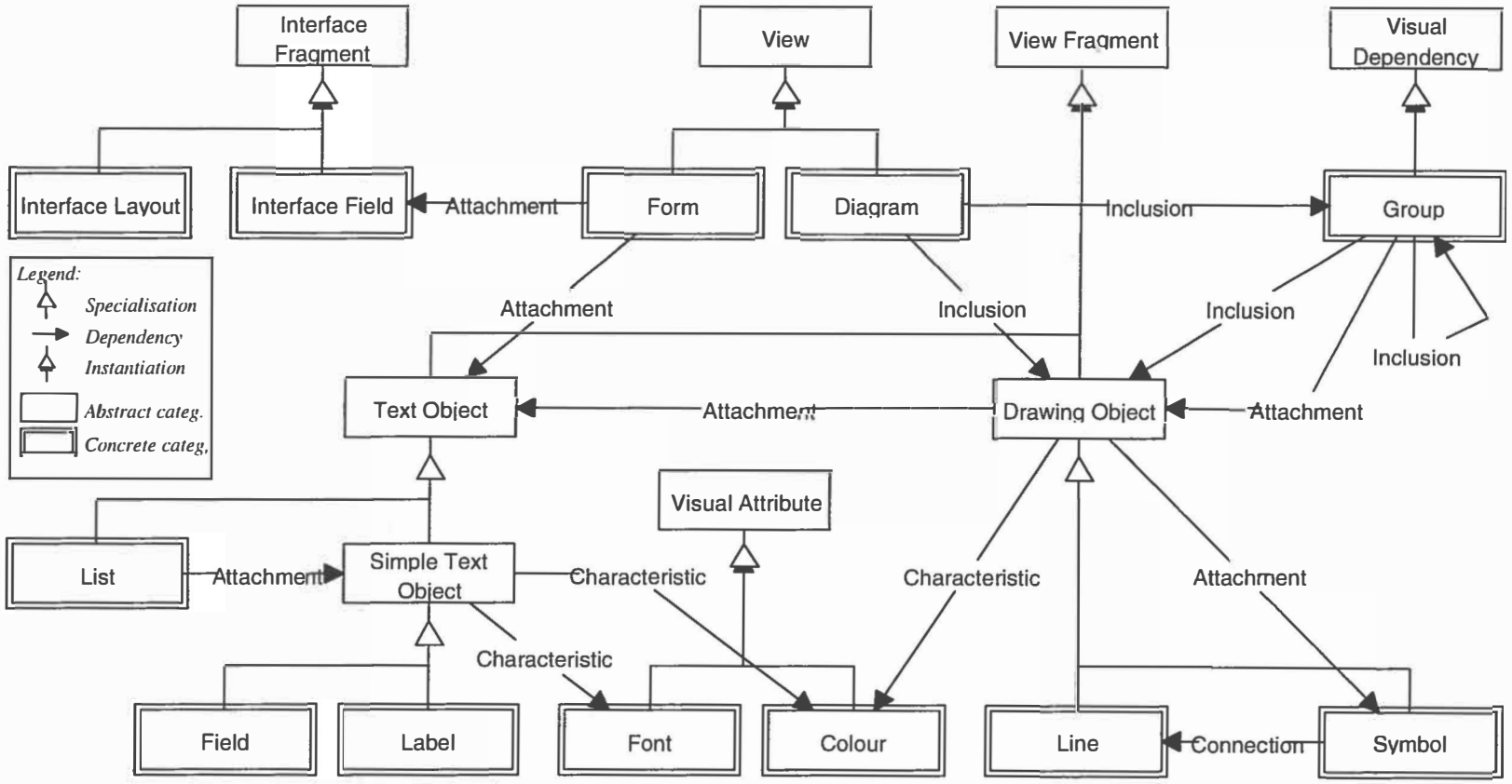

FIGURE 3 A model of an integrated diagram and form-based representation style (example). Linkages are possible among all construct excluding visual attributes (not shown).

dependencies shown in this model. First, we enumerate notational constructs that may appear within this representation style (Section 4.2.2), and then study the dependencies that may prevail between these notational constructs (Section 4.2.3). Thereafter, we integrate them to the conceptual ones (Section 4.2.4).

4.2.2 Notational Construct Categories: Examples

In our example, we use two view categories. Firstly, *diagrams* are views that show an arrangement of, and relations between, drawing objects. *Drawing objects* are graphical view fragments, and they include symbols, lines, and groups. *Symbols* are independent graphical 'nodes' that can be connected with lines. *Lines* are the 'edges' between symbols. *Groups* collect drawing objects together in a way that they can be manipulated as a single unit. Secondly, *forms* are views that are used for the manual input of data. Forms are composed of textual objects. *Textual objects* are textual view fragments, and they include fields, labels and lists. *Fields* enable the insertion and change of data values in a textual format. *Labels* show constant textual values.
There are also two interface fragment categories. Constructs in both categories are determined by constructs in the external system. Their difference is that *interface layouts* appear in the form of views whereas *interface fields* appear in the form of embedded fields. Both allow access to external data and media.

Two visual attribute categories, *font* and *colour*, are included. To be exact, a font combines several visual attributes: font type (e.g., Times, Courier), font size, and possibly font effects (e.g., bold, italic, underline). Colour could also be included in fonts, but since constructs with font (i.e., textual objects) are not the only constructs with colour, we regard it as an independent view attribute.

4.2.3 Notational Dependency Categories: Examples

The example employs all four main notational dependency categories. Firstly, *linkage dependencies* may appear between all notational constructs excluding visual attributes. Linkages visualise a guided change of focus. Putting a view on top of others, and centring a view on a certain view fragment, are perhaps the most common appearances of linkages.

Secondly, the example shows that *attachment dependencies* fix text objects and interface fields to forms, text objects and symbols to drawing objects, and drawing objects to groups. All operations (e.g., scaling) performed on a construct will affect all constructs involved in the attachment in some predefined manner.

Thirdly, the example uses *connection dependencies* between symbols and lines. Lines are regarded as drawing objects that have at least two end points, and one middle point. When a line is connected to a symbol, one of the ends is fixed to the symbol. Thereafter, all changes in the symbol's location on a diagram will change the location of this end whereas the other ends are not affected. Whether the middle point changes and how, is dependent on the line.

Fourthly, we use *characteristic dependencies* to give simple fields and labels their font and colour. Symbols and lines have only colour.

4.2.4 Integration to the Conceptual Categories: Examples

Before we continue our discussion of the integration of representation styles to conceptual categories, we point out two issues. Firstly, different families of notations may appear within one representation style. These families are formed when a representation style is related to conceptual categories. Let us consider the above-mentioned representation style discussed as an example. Some diagram-based notations could use symbols for process elements and lines for dependencies, whereas others could use lines for process elements and symbols for dependencies. Note that there might be no visual difference between these families except of what could be grasped of textual labels and fields, or specific pictures.

Secondly, the representation of complex information may benefit from simultaneous use of various notations. Representation independence should hence be supported so that one concept can be represented from different perspectives using different notations and even different representation styles without jeopardising the integrity of models. Representation independence distinguishes between a conceptualisation system (for composing a process model) and a representation system (for visualising it). Since these two systems are integrated, but autonomous, several different representations can be built for one conceptual process model, and yet be easily managed.

Integration is achieved by mappings between conceptual and notational constructs. A mapping specifies the rules that govern the use of a notational construct in relation with a conceptual one. At simplest, one might specify that when a certain type of process graph is represented as a certain type of diagram, all process elements of a certain type in that process graph will appear in the form of a certain kind of symbol.

In the following we illustrate an example of a conceptual model of notational process metamodels. This example is based on the representation style we specified above. We discuss this model in two parts. In the first part we concentrate on the diagram-based view of the model shown in figure 4. This view represents a family of diagram-based notations in which process elements are shown as symbols and internal dependencies as lines. In the second part we focus on the form-based view shown in figure 5. It represents a family of form-based notations in which entities are visualised as separate forms and their properties as the fields of these forms. In the figures, a mapping category is shown as a tuple of a notational category (above line) and a conceptual category (below line). Note that the 'direction' of a notational dependency is not necessarily the same as for the related conceptual dependency.

In diagram-based notations, process graphs are represented as diagrams, while process elements as symbols, and objectified dependencies as lines. Complex elements are represented as groups that combine a subset of symbols and lines by means of inclusion dependencies. Inclusion is related to

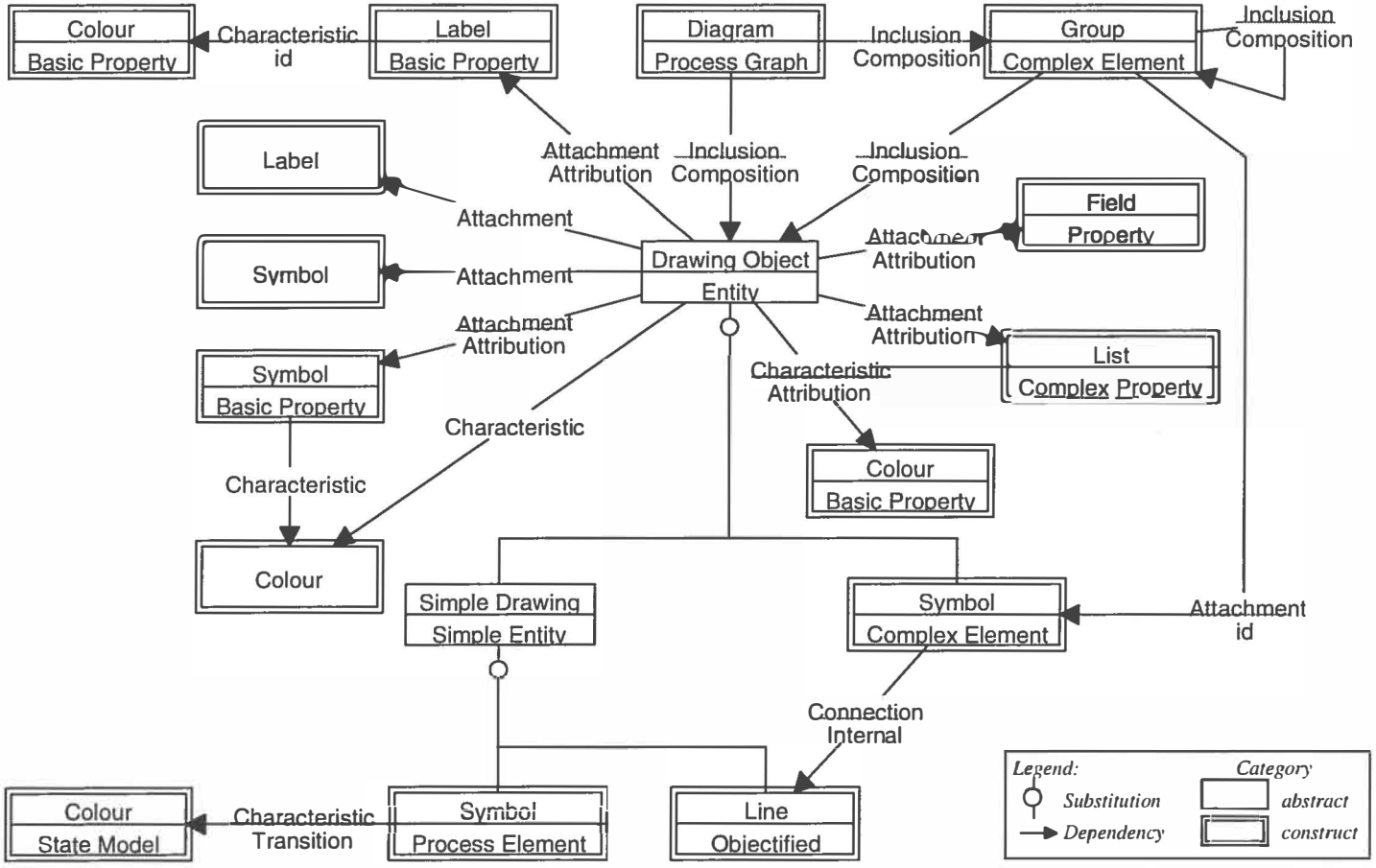

FIGURE 4 A conceptual model of notational process metamodels for diagram-based notations (example). Linkages can be created to represent association (not shown).

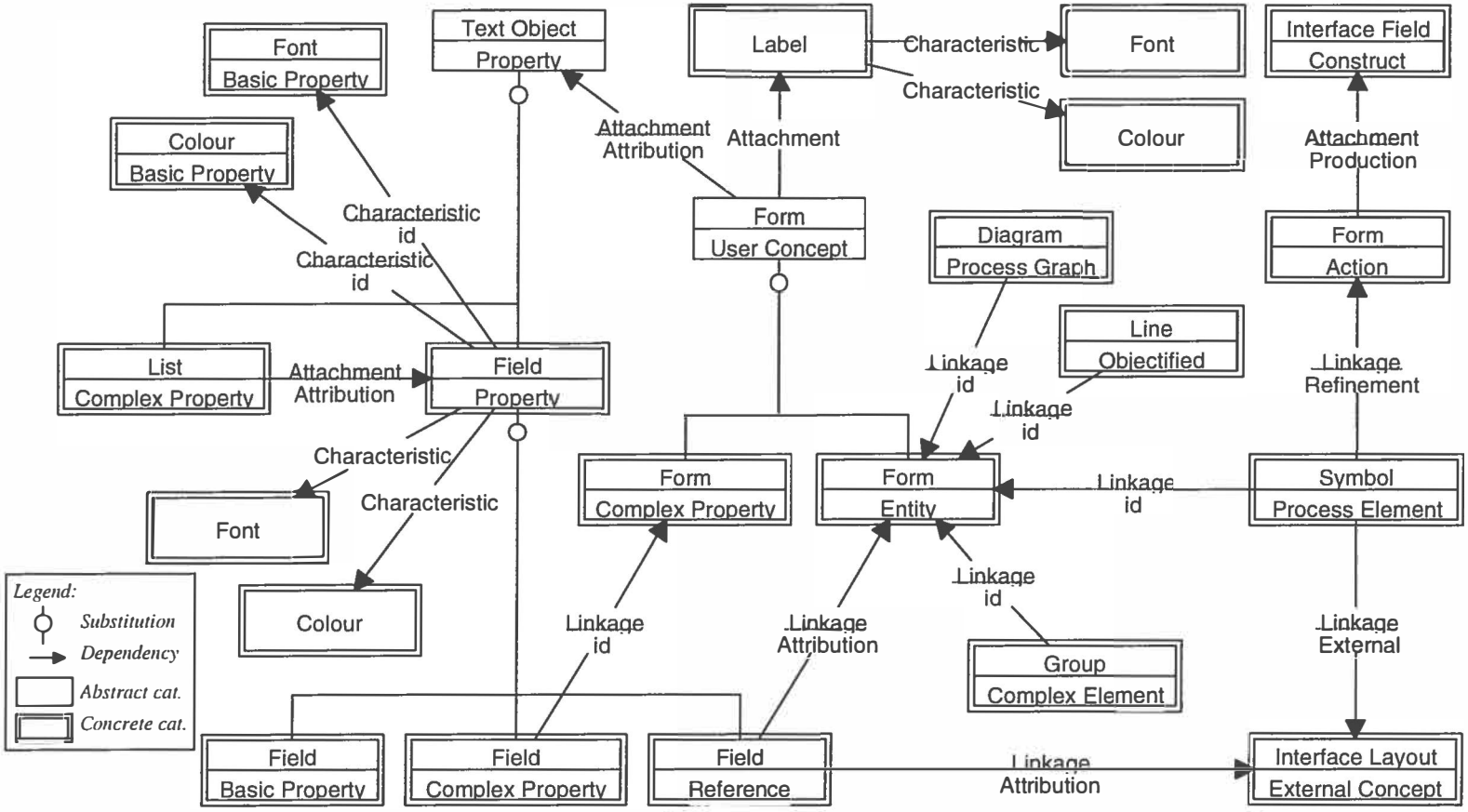

FIGURE 5 A conceptual model of notational process metamodels for form-based notations (example). Linkages can be created to represent association (not shown).

composition dependencies. Groups may attach symbols, such as boundaries. This is determined by concept identity. That is, both the group and the symbol are directly used as representations of the same complex element. They engage, however, in different kinds of dependencies. On one hand, a group has inclusion dependencies with symbols, lines and other groups, representing the composition of complex elements. Also, it has a linkage dependency to a form based on the concept identity (figure 5). On the other hand, the symbol has connection dependencies with lines representing the element's internal dependencies with objectified dependencies.

Internal dependencies are shown as connection dependencies. All symbols and lines may attach other, arbitrary symbols and labels, and have arbitrary colours. Furthermore, symbols representing process elements may have colour that represents state model transitions. In other words, the colours of symbols will change along with the change of process element states during model enactment. In case enactment should affect representation, such as changing colours, a notation must be connected to a semantics via the conceptual framework.

The properties of entities shown in diagrams may be represented both textually and symbolically. Textual representation is achieved by means of fields and lists attached to symbols and lines. Fields may represent any type of properties, whereas lists represent complex properties. Symbolical representation is achieved by special symbols and labels attached to symbols and lines, or by means of value-specific colours on symbols, lines, and attached labels. Only basic properties may be represented symbolically. Attribution is shown as attachment in case of fields, lists, symbols and labels, and as characteristic dependencies in case of colours. When arbitrary labels are related with colours specific to property values, labels and colours are related to properties and the characteristic dependency is based on concept identity.

In form-based notations, different entities are represented as separate forms. The forms provide an alternative view for diagrams and entities represented therein. All diagrams, groups, symbols, lines representing entities engage in a linkage dependency with a corresponding form.

The properties of the entities are represented as the fields of the forms. Basic properties are represented as fields with font and colour. Font and colour may also be value-specific in which case the characteristic is based on concept identity. Both complex properties and reference properties may be represented as the fields of forms. Such a field shown an arbitrarily constructed 'title', which is concatenated from the attributed concept according to specified rules. The attributed concept can be viewed separately. Entities are shown in forms, whereas external concepts are shown as interface layouts. Fields showing references have a linkage to forms representing entities or to interface layouts representing external concepts based on attribution. Complex properties may also be represented as lists attached to forms. Lists visualise complex properties as a collection of fields, each of which represents one of the attributed sub-properties. Besides fields, all forms may attach arbitrary labels with arbitrary font and colour.Actions are viewed as forms that contain interface fields for the representation of conceptual constructs. The attachment dependencies between

these forms and interface fields are related to production dependencies. Symbols that represent process elements engage in linkage dependencies (based on refinement dependencies) with forms. These symbols engage also in linkage dependencies with interface layouts. The linkage dependencies relate to external dependencies and the interface layouts relate to external concepts.

4.3 A Model of Semantic Process Metamodels

Since our objective is to design enactable process modelling languages, we focus here on the computational aspects of language semantics. This cannot be done without first discussing some technical matters of modelling and enactment.

A process support approach with customisable language semantics requires a generic process engine such as discussed in Section 2.3. The process engine needs to distinguish between the semantics of a language (according to which it interprets), the enactment mechanism (according to which it enacts), and the reflection mechanism (according to which it changes or allows the change of a process model). However, this does not entail that semantics and the generic enactment mechanism can be specified independently. The generic enactment mechanism determines a practical framework within which different semantics can be specified. Thereby it also determines the scope and generality of a process metamodelling approach. Such a framework is necessary before we can discuss a model of semantic process metamodels.

We outline six functional areas that encompass factors of managing automated enactment. These areas are shown in figure 6. Firstly, *management of progress* is the core area of enactment since it considers the enactment structure of process fragments. Enactment of process fragments can be approached from two points of view: lateral and hierarchical. Lateral management is targeted at the progress of individual execution threads and the relations between different, simultaneous execution threads. Hierarchical management is targeted at decomposition and refinement: choosing alternative process fragments and managing execution between hierarchical levels. Note that execution threads are not necessarily bound to the limits of one process fragment. They may also advance across sub-fragments within different fragments even at different abstraction levels.

Secondly, process fragments may have a set of generic properties with variable values. These include descriptive values such as those of a name, calculated values such as those of duration, constricting values such as those of resource availability, etc. *Management of variable values* concerns the management of generic properties both regarding their value changes and the effects of these changes on process enactment.

Thirdly, *management of execution* is the area of controlling the operations executed on some artefacts. The level of control may vary according to whether it concerns 'white-box' (high control) or 'black-box' (low control) operations. The operations can be either internal, in which case they operate on some aspect of the process model itself, or external, in which case they operate with some external tools. Internal operations are always 'white-box' operations.

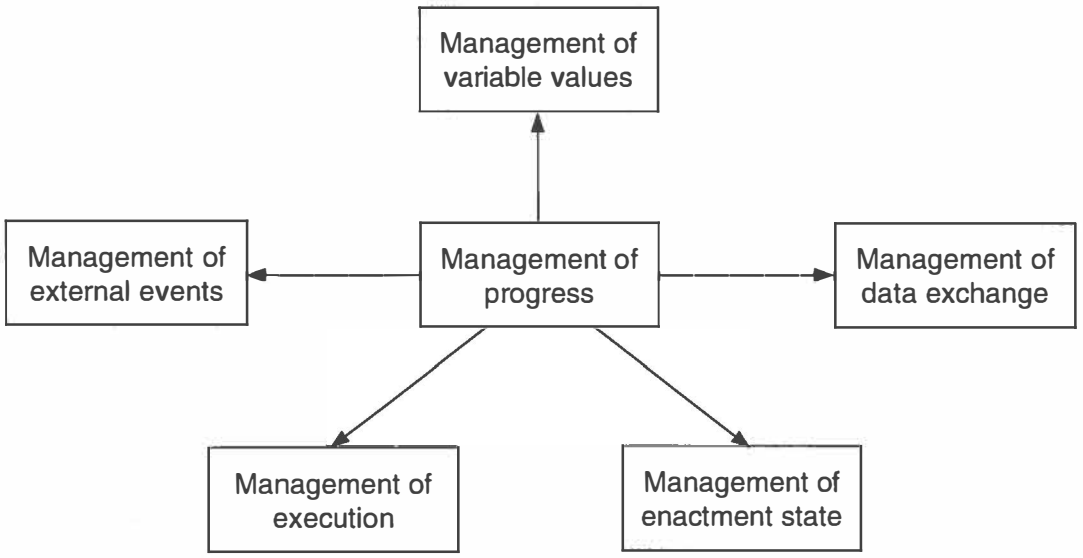

FIGURE 6 Areas of managing automated enactment.

Fourthly, an enacted process fragment has an enactment state that changes according to enactment. *Management of enactment state* concerns the evolution of these states: detecting advancements (or regression) in enactment, evaluating the corresponding state changes, and triggering new advancements (or regression).

Fifthly, process enactment is not narrowly concerned with behaviour and events within the enactment system but also with events external to it. Therefore, there has to be some means to integrate the enactment system with the external systems within which these events arise. *Management of external events* provides enactment control for detecting and informing such events.

Fifthly, process enactment is not narrowly concerned with behaviour and events within the enactment system but also with events external to it. Therefore, there has to be some means to integrate the enactment system with the external systems within which these events arise. *Management of external events* provides enactment control for detecting and informing such events.

Sixthly, process enactment references, uses and modifies also data that is stored external to the enactment system. It would benefit from mechanisms that can access external data and transform it between different formats (both from and to external ones) so that it can be directly used and manipulated in the enactment system. *Management of data exchange* provides enactment control for such import and export operations.

The above areas form a framework within which we develop a conceptual model of semantic metamodels. Such a model classifies different kinds of semantic constructs and dependencies that appear in semantic process metamodels. These semantic categories are then integrated with conceptual categories. The categories are shown in figure 7. In the following, we discuss these categories (in Sections 4.3.1 and 4.3.2) and integrate them with the conceptual ones (in Section 4.3.3).

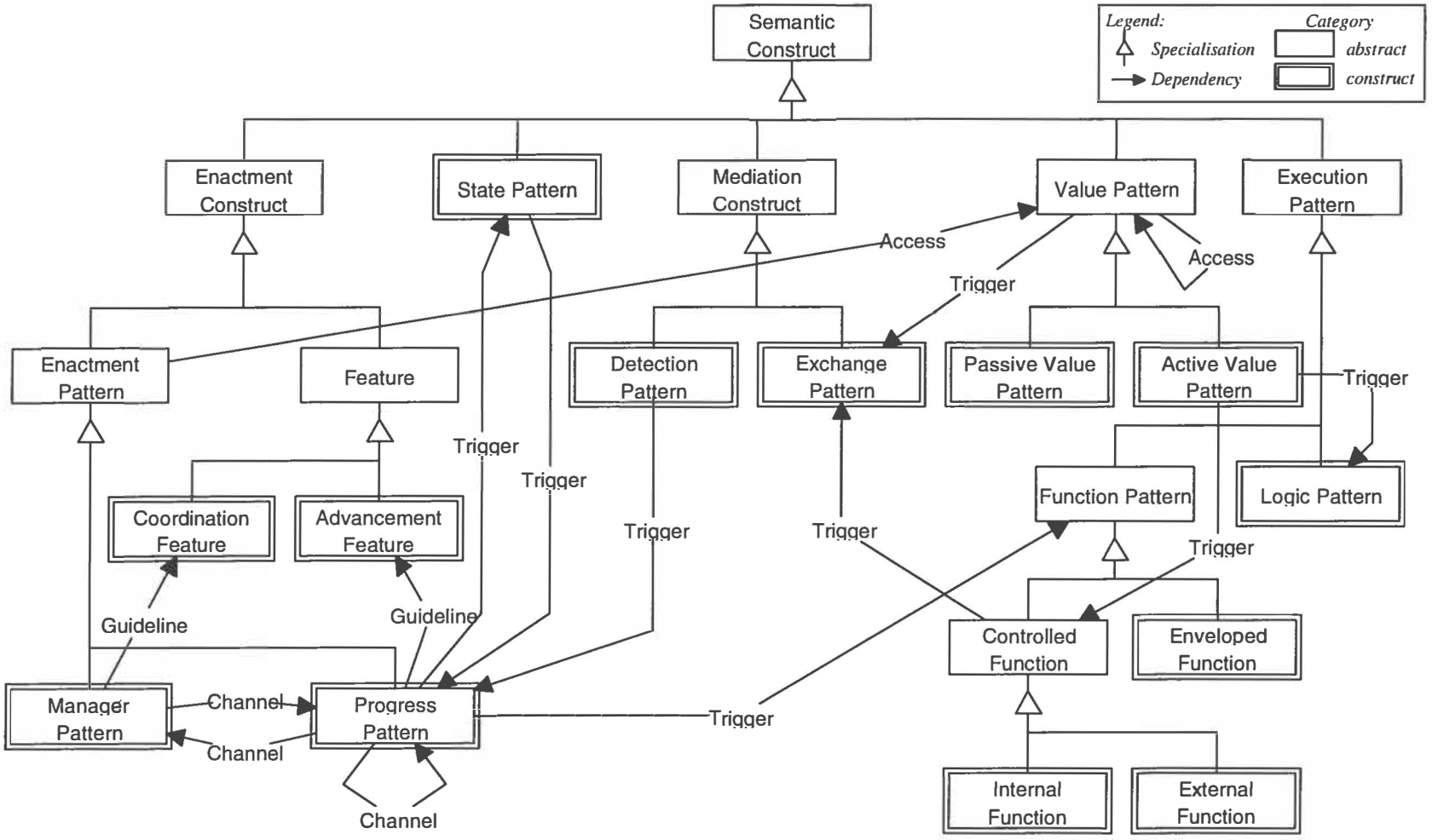

FIGURE 7 A model of semantic construct and dependency categories.

4.3.1 Semantic construct categories

Each semantic category addresses a single semantic aspect of enactable process modelling languages. Firstly, semantic issues that relate to enactment progress are specified with enactment patterns. *Progress patterns* are constructs that specify the advancement (and regression) of execution threads along a series of lateral and hierarchical execution channels. *Manager patterns* are constructs that control the interplay of several successive or simultaneous progress patterns. Enactment patterns compose a set of features that act as their enactment guidelines. Each feature determines a single generic aspect of enactment: either of advancement or co-ordination of concurrent execution threads. Features are classified accordingly as *advancement features* and *co-ordination features*. Enactment patterns also act as junction points for handling control signals from other areas of enactment.

Secondly, for the management of variable values, value patterns are used to specify the formation and the role of property values in process enactment. Some values are merely descriptive ones and they are entered into the system manually, whereas calculated or inferred ones are automatically computed. The former are handled by *passive value patterns*, whereas the latter are by *active value patterns*. Constricting values may be manually entered or automatically computed. Passive value patterns may be constricting only based on value, whereas active value patterns may be constricting also based on the success of producing or using the value in an operation.

Thirdly, in the area of execution control, execution patterns are used to manage functional and logical operations. Firstly, *function patterns* specify the type of operation invocation and the mode of control during its execution. Controlled function patterns specify control for white-box operations. These include *internal function patterns* for reflective operations and *external function patterns* for non-reflective ones. Reflection may address both the structure of process fragments and the values of their properties. The latter is used, e.g., for token manipulation during the execution of Petri-net based process models. Also, reflection can be used for initiating exceptional state changes. *Enveloped function patterns* specify control for externally executed 'black-box' operations. The true controllability of such operations is very low, which means that some manual input may be required to track their progress. Secondly, *logic patterns* specify compound logical operations.

Fourthly, for the management of enactment state changes, *state patterns* specify how the enactment state is changed in response to internal events. They specify rules according to which the enactment state changes. Such language constructs are explicit in Petri-net based languages, but similar mechanisms are also found implicit in the use of log files, or some other external medium. A support environment uses such mechanisms at least to check whether some step has been performed or not. External medium is needed when a language lacks constructs to represent state information. It is here important to understand the difference between a language and a language implementation. The semantics of enactment state evolution is part of language, whereas the actual mechanisms of

evolution concern language implementation. A language can be implemented in different ways.

Two construct categories concern information exchange between the enactment system and an external system. Firstly, semantics related to external event detection is specified with detection patterns. *Detection patterns* specify how events are raised and detected in an external system. Secondly, semantics related to data exchange is specified with exchange patterns. *Exchange patterns* specify a transformation schema for a mapping between the data format used in the enactment system and the data format used in an external system.

4.3.2 Semantic dependency categories

Two semantic dependency categories relate to progress management. First, *channel dependencies* specify the advancement of execution threads between enactment patterns. Channel dependencies can be lateral or hierarchical. Second, *guideline dependencies* appear between enactment patterns and features. The guidelines of progress patterns consist of advancement features, whereas manager patterns compose co-ordination features.

A category relating to the management of variable values is the category of *access dependencies*. They are contextual and govern the effect of variable values on enactment. Access dependencies usually appear between enactment patterns and value patterns, but they may also be used to compose value patterns into larger compounds.

The largest category of semantic dependencies is not specific to any particular area of enactment, but relates to interactions between different areas. *Trigger dependencies* specify executive relationships, i.e., triggering of execution. They appear between progress patterns and execution patterns, between detection patterns and progress patterns, between progress patterns and state patterns (and vice versa), between active value patterns and exchange patterns, as well as between active value patterns and controlled function patterns, or logic patterns.

4.3.3 Integration to the Conceptual Categories

When semantic categories are related with conceptual ones, a model of semantic process metamodels is attained. The conceptual model of semantic process metamodels shown in figure 8 relates semantic categories to the model of conceptual process metamodels developed in Section 4.1. The mapping categories are shown similarly as for notational categories. The name of a semantic category is shown above the name of the conceptual one. More than one semantic category can be related to a conceptual category.

The semantics of conceptual constructs are specified with semantic constructs. Firstly, the semantics of entities is determined by enactment patterns. Process elements and objectified dependencies are simple constructs and thus we specify their semantics using progress patterns. Since complex elements have partially similar type of semantics as process elements, they are also related to

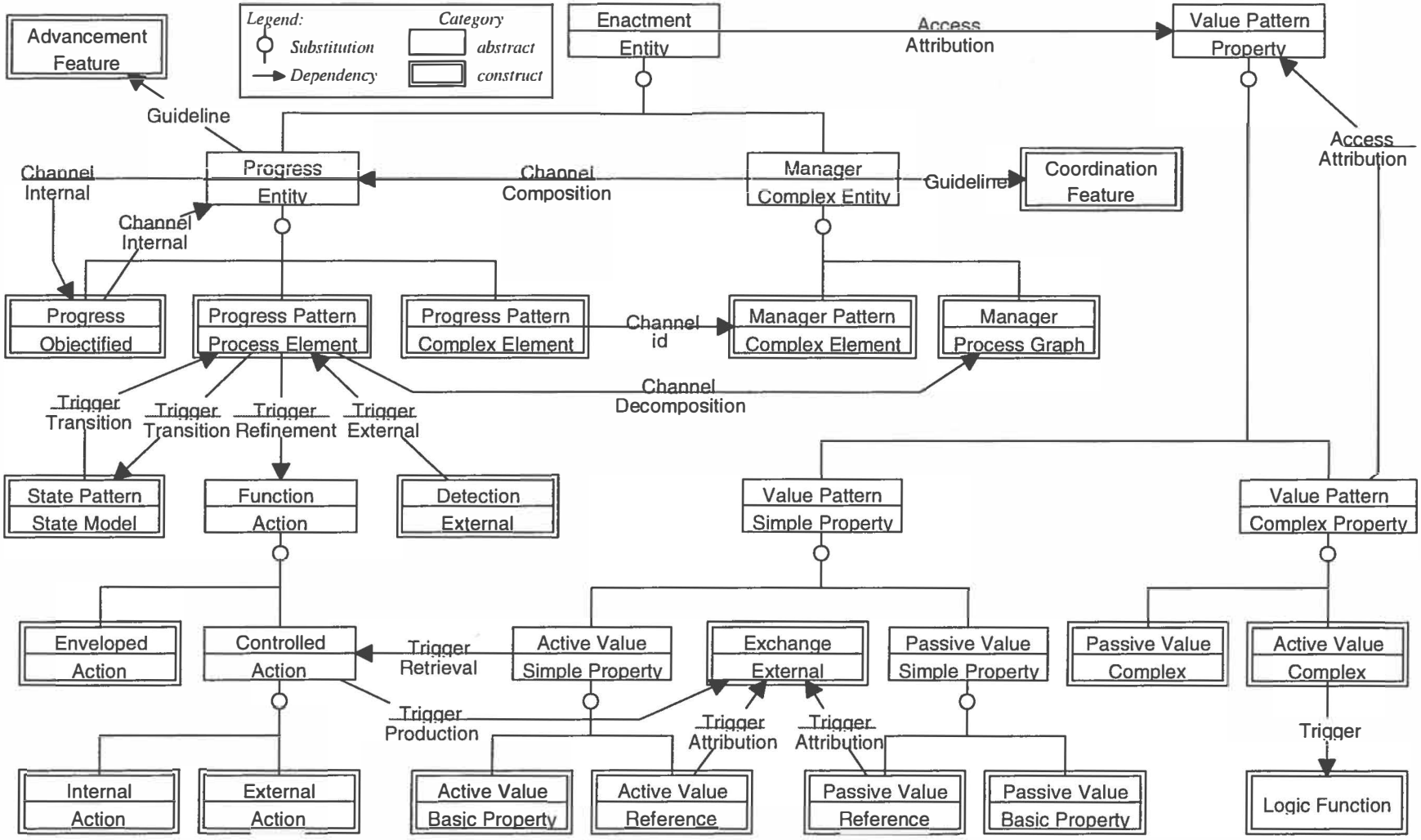

FIGURE 8 A conceptual model of semantic process metamodels.

progress patterns. However, since they are also compounds of other entities, their semantics is extended using manager patterns. Thus, complex elements have simultaneously two kinds of semantic pattern. The semantics of process graphs is specified with manager patterns. Features have no counterpart in conceptual categories.

Secondly, semantics of properties is determined by value patterns. Basic properties 'hold' an immediate value, whereas for reference properties they 'hold' a reference to an autonomous object. A complex property composes a set of other properties and its 'value' is the set of values composed from those properties. The value of a complex property can not be calculated (even though its representation could be). Calculated values are determined through action execution and 'stored' in a property. Any kind of property can act as a constricting property. The success of producing or using a value is determined from action execution. Also complex properties may be related with active value patterns to make them constricting, but they use logic patterns to evaluate the constraint.

Thirdly, the semantics of actions is defined through function patterns. While an action type specifies the conceptual schema of a class of operations, the related execution pattern determines its formal semantics. Logic patterns do not have a conceptual counterpart.

Fourthly, the semantics of state models is determined by state patterns. While a state model specifies an enactment life-cycle, a state pattern specifies how this life-cycle is enacted.

Fifthly, the semantics of external concepts is specified with detection patterns or exchange patterns depending on the role they have in enactment. External concepts are related with the former when they refer to external events, and with the latter when they refer to external data.

Conceptual dependencies are related to semantic dependencies. Firstly, the semantics of dependencies between entities is determined by channel dependencies. Thus, channels between progress patterns relate to internal dependencies. Channel dependencies between manager patterns and progress patterns relate to composition dependencies, and channel dependencies between progress patterns and manager patterns relate to decomposition dependencies. Since complex elements have both a process pattern and a manager pattern, the dependency between patterns is based on the identity of the complex element. Guidelines do not have conceptual counterparts.

Secondly, the semantics of attribution dependencies is determined by access dependencies. Access dependencies collect the values of properties to a compound construct (i.e., an entity or a complex property).

Thirdly, trigger dependencies can be related to different conceptual dependency categories. With refinement and evaluation dependencies, they evoke operation execution. With attribution and production dependencies, they transfer external data into the enactment system and internal data into an external system. With external dependencies, they mediate information of external events into the enactment system, whereas with transition dependencies they transfer information of internal events into the state engine and vice versa.

5 Towards a Model of Technique-based Process Metamodels

Finally, we outline a further extension of the approach towards technique-based process metamodelling. A *technique-based process metamodel* includes a component equivalent to a language-based process metamodel, but extends it with another component, an operational model. An *operational model* captures the operational semantics of a process modelling technique. The model codifies procedures of manipulating different components of process models. In a tool environment, an operational model can be used to specify and control operations available for a tool user (e.g., as menu options).

Operations create or abide to dependencies between components and thus operation categories are related to these dependencies. We choose to construct operations from a set of generic operations. Each of the generic operations is potentially available in a particular generic model context. The context of an operation is formed of the component at which it is applicable. In operational modelling, the generic operations are merged into atomically performed composites that are provided as operations available for users.

In the following we discuss two types of operations. Firstly, model and tool operations (Section 5.1) relate to model and tool components. They comprise the core operations of process modelling. Secondly, support extensions (Section 5.2) provide operations that combine modelling with other support areas. For the latter, we do not consider operation categories in detail but merely outline some directions. Anyhow, we expect that many of the operation categories are common to the modelling area, although some new operation categories are also introduced.

5.1 Model and Tool Operations

Model and tool operations relate to dependencies between model components or model and tool components. *Model components* include conceptual, representation, and interpretation components, which are instantiated from conceptual, notational, and semantic metamodels, respectively. *Tool components* refer to components, the manipulation of which relates only to model presentation, such as viewing or printing.

Sample categories of modelling operations are shown in table 2. Examples are added where necessary. Firstly, existence and containment operations apply to model components. *Existence operations* (create, delete) are available where components could be created or deleted. Note that creation may use a 'template', as in copy operations. *Containment operations* (add, remove) are available where a component could have sub-components. Containment operations add or remove existing sub-components to or from a given component.

TABLE 2 Sample categories of model and tool operations.

Operation category	Model			Tool	Examples
	C	R	I		
Existence (create, delete)	*	*	*		
Containment (add, remove)	*	*	*		
Progression (start, finish)			*		
Intervention (suppress, resume)			*		
Selection (select, deselect)	*	*	*	*	
Variable change (edit)	*	*	*	*	edit string, align, move, scale, set state, zoom, set grid
Transformation (encode, decode)	*	*	*	*	import, export, print, report
Reversion (undo, redo)	*	*	*	*	
Interface (open, close)				*	
Appearance (show, hide)				*	show/hide symbol, show/hide grid

Legend: C = Conceptual componentR = Representation component I = Interpretation component

Secondly, progression and intervention operations apply to semantic components. *Progression operations* (start, finish) are available where the execution of a given component could be started or finished. *Intervention operations* (suppress, resume) are available where execution could be suppressed or resumed. Most progression operations are automatically performed by the process engine.

Thirdly, operations for selection, variable change, transformation and reversion may apply to model or tool components. *Selection operations* (select, deselect) are available when a user could change the current context of operation (i.e., select or deselect a component). *Variable change operations* (edit) are available where it is more feasible to manipulate the sub-components of a component within a single operation, than to use existence and containment operations (e.g., edit string, align, move, scale, set state, zoom, set grid). *Transformation operations* (encode, decode) are available where model information in one format could be transformed into another. Examples include operations for importing, exporting, printing and reporting. *Reversion operations* (undo, redo) are available where a certain series of operations could be undone (cancelled), or redone atomically.

Fifthly, interface and appearance operations apply to tool components. *Interface operations* (open, close) are available where the contents of sub-components could be shown in another view or this view could be closed. *Appearance operations* (show, hide) are available where sub-components could be visible or invisible within the current view.

5.2 Support Extensions

Model and tool operations constitute the core of modelling-related operations, but they are still only some of several available. There are other relevant areas such as access management, change management, version and configuration management, and component management. Each area has a viewpoint that differs from the one used in modelling. They introduce additional concepts and dependencies that are not used to represent systems development processes but different operational aspects of process modelling itself. Therefore, we do not regard them as process modelling constructs but, instead, as operational modelling constructs.

We distinguish among three operational levels. Firstly, modelling support operations are provided on the *technique-level*. This level includes, e.g., change management, traceability (cf. Pohl & Weidenhaupt, 1997) and decision support (cf. Si-Said et al., 1996) for process modelling. Modelling support operations automate some technique-specific operation sequences to support the use of the technique. Secondly, component management and reuse operations are interfaced to process modelling on the *component-level*. This support may be customisable beyond the mere ordering of operations since component management and reuse involves a consideration of human processes. However, the customisation should be performed separately to reduce the complexity of operational modelling. Thirdly, such areas as access, version and configuration management operate on the *technical base-level*. The nature of these areas is overwhelmingly technical and thus we do not expect substantial benefit from their customisability. However, it is necessary to make operations also at this level available in operational modelling.

The above areas are the most obvious extensions to modelling, but also others may be introduced.

6 Conclusions

This article is an off-spring of a study conducted over the last five years (Koskinen and Marttiin, 1997; Koskinen, 1999). The study has gone through several iterated cycles of theory building, prototype development, and experimentation, during which the proposed approach has gradually evolved into the presented form.

We have considered conceptual foundations of process metamodelling, and developed a structural view of process metamodels based on a discussion of the linguistic base of process modelling languages and techniques. Thereafter, we have used this structure to construct a model of process metamodels that includes three aspects for the specification of process modelling languages: conceptual, notational and semantic process metamodels. This model can be further supplemented with a model of operational models to extend the approach for the specification of process modelling techniques. The

model of process metamodels is also a conceptual model of a process meta-metamodel that can be used as a basis of a customisable process support architecture.

A major limitation in this article is the lack of a discussion of abstraction mechanisms in process modelling and process metamodelling languages. Our experiments on process metamodelling have shown that different forms of abstraction, such as metamodel patterns, are indispensable in practice.

We conclude that conceptual systematisation is necessary both for process modelling research and practice, but it should not be conducted as standardisation of languages per se. In theoretical and practical development of language design, this is especially important. Moreover, the lack of common, integrated model makes it impossible to compare and evaluate languages systematically.

This work forms a cornerstone for the research on process modelling language design and process support customisation, particularly in metaCASE. A comprehensive process meta-metamodel is necessary as a basis of a metaCASE process support architecture, where extensive customisability of methods and process approaches is expected. Besides metaCASE research, we expect this work to benefit anyone who is interested in the conceptual design, or comparison of process modelling languages.

References

Baldinger, K. 1980. Semantic Theory. Towards a Modern Semantics. Oxford: Basil Blackwell.

Balzer, R. & Narayanaswamy, K. 1993. Mechanisms for generic process support. In D. Notkin (Ed.) Proceedings of the 1st ACM SIGSOFT Symposium on the Foundations of Software Engineering. Special Issue of Software Engineering Notes, 18, 5, 21-32.

Bandinelli, S., Fuggetta, A. & Ghezzi, C. 1993. Software Process Model Evolution in the SPADE Environment. IEEE Transactions on Software Engineering, 19, 12, 1128-1144.

Bergheim, G., Sandersen, E. & Solvberg, A. 1989. A taxonomy of concepts for the science of information systems. In E.D. Falkenberg & P. Lindgreen (Eds.) Information System Concepts: an In-Depth Analysis. Amsterdam: Elsevier Science Publishers, 269-321.

Braun, H., Hesse, W., Andelfinder, U., Kittlaus, H.-B. And Scheschonk, G. 1999. Conceptions are social constructs – Towards a solid foundation of the FRISCO approach. In: Information Systems Concepts: An Integrated Discipline Emerging. Proceedings of the IFIP WG8.1 International Conference ISCO-4. The Netherlands, 20-22 September 1999.

Chen, M. & Norman, R.J. 1992. A Framework for Integrated CASE. IEEE Software, March, 18-22.

Conradi, R., Fernström, C., Fuggetta, A. & Snowdon, R. 1992. Towards a Reference Framework for Process Concepts. In J.-C. Derniame (Ed.) Software Process Technology, EWSPT'92, LNCS 635. Berlin: Springer-Verlag, 3-17.

Conradi, R., Fernström, C. & Fuggetta, A. 1993. A Conceptual Framework for Evolving Software Processes. ACM SIGSOFT Software Engineering Notes, 18, 4, 26-35.

Conradi, R. & Jaccheri, M.L. 1993. Customization and Evolution of Process Models in EPOS. In N. Prakash, C. Rolland & B. Pernici (Eds.) Information System Development Process. Amsterdam: Elsevier Science Publishers, 23-39.

Conradi, R. & Liu, Ch. 1995. Process Modelling Languages: One or Many? In W. Schäfer (Ed.) Software Process Technology, EWSPT'95, LNCS 913. Berlin: Springer-Verlag, 98-118.

Davis, R., Shrobe, H. & Szolovits, P. 1993. What Is a Knowledge Representation? AI Magazine, Spring 1993.

Dowson, M. 1987b. Iteration in the software process. In Proceedings of the 9th International Conference on Software Engineering. Washington D.C.: Computer Society of the IEEE, 36-39.

Dowson, M. & Fernström, C. 1994. Towards Requirements for Enactment Mechanisms. In B. Warboys (Ed.) Software Process Technology, EWSPT'94, LNCS 772. Berlin: Springer-Verlag, 90-106.

Falkenberg, E., Hesse, W., Lindgreen, P., Nilsson, B.E., Oei., J.L.H., Rolland, C., Stamper, R.K., van Assche, F.J.M, Verrijn-Stuart, A.A. and Voss, K. 1998. FRISCO - A Framework of Information System Concepts - The FRISCO Report. IFIP WG 8.1 Task Group FRISCO.

Feiler, P.H. & Humphrey, W.S. 1993. Software Process Development and Enactment: Concepts and Definitions. In L. Osterweil (Ed.) Proceedings of the 2nd International Conference on the Software Process. Los Alamitos: IEEE Computer Society Press, 28-39.

Finkelstein, A., Kramer, J. & Nuseibeh, B. 1994. Software Process Modelling and Technology. New York: Wiley.

Fodor, J. D. 1977. Semantics: Theories of Meaning in Generative Grammar. The Language & Thought Series. Sussex: The Harvester Press.

Hardy, W.G. 1978. Language, Thought, and Experience. A Tapestry of the Dimensions of Meaning. Baltimore: University Park Press.

Jarke, M. & Rose, T. 1992. Specification Management with CAD^{o}. In P. Loucopoulos & R. Zicari (Eds.) Conceptual Modeling, Databases, and CASE. New York: Wiley, 489-505.

Jarke, M., Pohl, K., Rolland, C. & Schmitt, J.-R. 1994. Experience-Based Method Evaluation and Improvement: A process modeling approach. In T.W. Olle & A.A. Verrijn-Stuart (Eds.) Proceedings of the IFIP WG8.1 Working Conference CRIS´94. Amsterdam: North-Holland, 1-27.

Jarke, M., Pohl, K., Weidenhaupt, K., Lyytinen, K., Marttiin, P., Tolvanen, J.-P. & Papazoglou, M. 1998. Meta Modeling: A Formal Basis for Interoperability and Adaptability. In B. Krämer & M. Papazoglou (Eds.), Information Systems Interoperability, John Wiley Research Science Press, 229-263.

Joeris, G. 1997. Change Management Needs Integrated Process and Configuration Management. In M. Jazayeri & H. Schauer (Eds.) Software Engineering - ESEC-FSE '97, LNCS 1301. Berlin: Springer-Verlag, 125-141.

Kaiser, G.E. & Ben-Shaul, I.Z. 1993. Process Evolution in the Marvel Environment. In W. Schaefer (Ed.) Proceedings of the 8th International Software Process Workshop. Los Alamitos: IEEE Computer Society Press, 104-106.

Kaiser, G.E., Ben-Shaul, I.Z., Popovich, S.S. & Dossick, S.E. 1996. A Metalinguistic Approach to Process Enactment Extensibility. In W. Schaefer (Ed.) Proceedings of the 4th International Conference on the Software Process. Los Alamitos: IEEE Computer Society Press, 90-101.

Karrer, A.S. & Scacchi, W. 1993. Meta-environments for software production. International Journal of Software Engineering and Knowledge Engineering, 3, 1, 139-162.

Kelly, S., Lyytinen, K. & Rossi, M. 1996. METAEDIT+ — A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In P. Constantopoulos, J. Mylopoulos & Y. Vassiliou (Eds.) Advanced Information Systems Engineering, LNCS 1080. Berlin: Springer-Verlag, 1-21.

Koskinen, M. 1999. A Metamodelling Approach to Process Concept Customisation and Enactability in MetaCASE. University of Jyväskylä. Computer Science and Information Systems Reports, Technical Reports TR-20. Licentiate thesis. University of Jyväskylä, Finland.

Koskinen, M. & Marttiin, P. 1997. Process Support in MetaCASE: Implementing the Conceptual Basis for Enactable Process Models in MetaEdit+. In J. Ebert & C. Lewerentz (Eds.) Software Engineering Environments. Los Alamitos: IEEE Computer Society Press, 110-123.

Lehman, M.M. 1987. Process Models, Process Programs, Programming Support. In Proceedings of the 9th International Conference on Software Engineering. Washington D.C.: Computer Society of the IEEE, 14-16.

Lonchamp, J. 1993. A structured conceptual and terminological framework for software process engineering. In L. Osterweil (Ed.) Proceedings of the 2nd International Conference on the Software Process. Los Alamitos: IEEE Computer Society Press, 41-53.

Madhavji, N.H. 1992. Environment Evolution: The Prism Model of Changes. IEEE Transactions on Software Engineering, 18, 5, 380-392.

Marttiin, P., Lyytinen, K., Rossi M., Tahvanainen V.-P., Smolander K. & Tolvanen, J.-P. 1995. Modeling Requirements for Future CASE: modeling issues and architectural considerations. Information Resource Management Journal, 8, 1, 15-25.

Marttiin, P., Rossi, M., Tahvanainen, V.-P. & Lyytinen, K. 1993. A comparative review of CASE Shells: a preliminary framework and research outcomes. Information and Management, 25, 11-31.

Meyer, B. 1990. Introduction to the Theory of Programming Languages. New York: Prentice Hall.

Mi, P. & Scacchi, W. 1990. A Knowledge-Based Environment for Modeling and Simulating Software Engineering Processes. IEEE Transactions on Knowledge and Data Engineering, 2, 3, 283-294.

Mi, P. & Scacchi, W. 1991. Modeling Articulation Work in Software Engineering Processes. In M. Dowson (Ed.) Proceedings of the 1st International Conference on the Software Process. Los Alamitos: IEEE Computer Society Press, 188-201.

Mi, P. & Scacchi, W. 1996. A Meta-Model for Formulating Knowledge-Based Models of Software Development. Decision Support Systems, 17, 3, 313-330.

Mili, H., Pachet, F., Benyahia, I. & Eddy, F. 1995. Metamodelling in OO: OOPSLA'95 Workshop Summary. In Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages and Applications. ACM SIGPLAN Notices 30, 10, 105-110.

Nissen, H., Jeusfeld, M., Jarke, M., Zemanek, G. & Huber, H. 1996. Managing multiple requirements perspectives with metamodels. IEEE Software, March, 37-48

Phalp, K. & Shepperd, M. 1994. A Pragmatic Approach to Process Modelling, In B. Warboys (Ed.) Software Process Technology, EWSPT'94, LNCS 772. Springer-Verlag, 65-68.

Pohl, K. & Weidenhaupt, K. 1997. A Contextual Approach for Process-Integrated Tools. In M. Jazayeri & H. Schauer (Eds.) Software Engineering - ESEC-FSE '97, LNCS 1301. Berlin: Springer-Verlag, 176-192.

Rolland, C., Souveyet, C. & Moreno, M. 1995. An approach of defining ways-of-working. Information Systems, 20, 4, 337-359.

Rossi, S. & Sillander, T. 1998a. A Software Process Modelling Quest for Fundamental Principles. In R. Walter & J. Baets (Eds.) Proceedings of the 6th European Conference on Information Systems (ECIS). Euro-Arab Management School, Spain, 557-570.

Si-Said, S., Rolland, C. & Grosz, G. 1996. MENTOR: A Computer Aided Requirements Engineering Environment. In P. Constantopoulos, J. Mylopoulos & Y. Vassiliou (Eds.) Advanced Information Systems Engineering, LNCS 1080. Berlin: Springer-Verlag, 22-43.

Sorenson, P.G., Tremblay, J-P. & McAllister, A.J. 1988. The Metaview system for many specification environments. IEEE Software, 30, 3, 30-38.

Sutton, S.M., Tarr, P.L. & Osterweil, L.J. 1995. An Analysis of Process Languages. University of Massachusetts, Department of Computer Science. CMPSCI Technical Report 95-78. University of Massachusetts.

Sutton, S. & Osterweil, L. 1997. The Design of a Next Generation Process Language. In M. Jazayeri, H. Schauer (Eds.) Software Engineering - ESEC-FSE'97, LNCS 1031. Berlin: Springer-Verlag, 142-158.

Taivalsaari, A. 1993. A Critical View of Inheritance and Reusability in Object-oriented Programming. University of Jyväskylä. Jyväskylä Studies in Computer Science, Economics and Statistics 23. Ph.D. Thesis.

Tolvanen, J.-P. & Lyytinen, K. 1993. Flexible method adaptation in CASE - the metamodeling approach. Scandinavian Journal of Information Systems, 5, 51-77.

Verhoef, T.F. & ter Hofstedte, A.H.M. 1995. Feasibility of Flexible Information Modelling Support. In J. Iivari, K. Lyytinen & M. Rossi (Eds.) Advanced Information Systems Engineering, LNCS 932. Berlin: Springer-Verlag, 5-20.

PART III: THE CPME PROTOTYPE

5 DEVELOPING A CUSTOMISABLE PROCESS MODELLING ENVIRONMENT: LESSONS LEARNT AND FUTURE PROSPECTS

Foreword

This paper has been published in 1998, and thus does not reflect any later changes. Hence, there is a difference between the domain classification used in this paper and the classification made later in Chapter 7.

In this paper, the process metamodelling and process modelling domains are considered as different domains, whereas in Chapter 7 they will appear in the same domain, "method definition domain", but as different subsystems. Process metamodelling will be placed in "technique specification system" (for process modelling techniques), and process modelling will be placed in process modelling system. Further, process enactment will appear in the "process enactment system" in the "method enactment domain", and process performance will be distributed in the "IS/Software specification system" and the "development system", both in the "performance domain".

Koskinen M. & Marttiin P. "Developing a Customizable Process Modelling Environment: Lessons Learnt and Future Prospects", in V. Gruhn (ed.) Software Process Technology (EWSPT '98), LNCS #1487, Springer-Verlag, Berlin, 1998, pp. 13-27.

© 1998 Springer-Verlag. Reprinted, with permission, from Software Process Technology, Springer-Verlag.

https://doi.org/10.1007/3-540-64956-5_2

6 PROCESS SUPPORT IN METACASE: IMPLEMENTING THE CONCEPTUAL BASIS FOR ENACTABLE PROCESS MODELS IN METAEDIT+

Foreword

This paper has been published in 1997, and is thus the earliest one in the collection of papers. There are two issues that I consider to need further clarification in regard to the other papers.

First, the architecture illustrated on page 139 can be mapped to the architecture presented in Chapter 7 in the following way. The method level corresponds with the performance domain. IS models are created in the IS/Software specification system (7), and the process is carried out in the development system (8). Definition of metadata models on the method definition level is related to the technique specification system for system modelling techniques (1b), and definition of process models to the process modelling system. PML definition on the method definition schema level relates to the technique specification system for system modelling techniques.

Second, the metatypes illustrated on pages 141-142 precede the version discussed in Chapter 3. The metatypes are Process Element, Action, Relationship, Role, Graph, and Property. The first two appear as such also in Chapter 3, and Graph renamed as Process Graph to distinguish it from the GOPRR metatype Graph. Property is later divided into Basic Property, Reference Property, and Complex Property. Relationship and Role have been merged into Objectified Dependency. Complex Element, State Model, and External Concept do not yet appear as independent metatypes. Other changes will be discussed in Chapter 7.

Unfortunately, reproduction of the paper in a shrunk in-press format makes small-size figures blurred. For the convenience of the reader, the figures are reprinted in a greater size on pages 150-154.

Koskinen, M. & Marttiin, P. "Process Support in MetaCASE: Implementing the Conceptual Basis for Enactable Process Models in MetaEdit+", in J. Ebert, C. Lewerentz (eds.) *Proceedings of the 8th Software Engineering Environments,* IEEE Computer Society Press, 1997, pp. 110-123.

© 1997 IEEE. Reprinted, with permission, from Proceedings of the 8th Conference on Software Engineering Environments; Cottbus, Germany, April 8-9; pp. 110-123.

https://doi.org/10.1109/SEE.1997.591823

PART IV: ASSESSMENT

7 A GENERIC PROCESS MODELLING AND ENACTMENT SYSTEM: IMPLEMENTATION AND ASSESSMENT

Koskinen, M. & Marttiin, P. "A Generic Process Modelling and Enactment System: Implementation and Assessment".

A shorter version of this paper will be submitted to IEEE Transactions on Software Engineering for possible publication.

© 2000 IEEE. Printed with permission.

A Generic Process Modelling and Enactment System: Implementation and Assessment

Minna Koskinen
University of Jyväskylä

Pentti Marttiin
Nokia Research Center

Abstract

An increasing awareness of the benefits of process support in metaCASE environments has taken place in recent years. Most of the issues of interest are directly derived from existing software process research. Customisation of languages and techniques for process modelling, in contrast, is quite a unique theme to metaCASE research. It is part of a larger research effort aiming at developing comprehensive customisable method support environments. Today, research on metaCASE process support is diverse and scattered. There are no general architectures that would show direction for a unified body of relevant research. This article contributes to this end. We have developed an assessment framework for customisable method support environments. The framework contains an extensive set of criteria against which to evaluate an architecture of a method support environment. We present a prototypical, generic process modelling and enactment system implemented for a metaCASE environment. We assess the prototype against the developed criteria, and identify themes for future research on customisable method support environments. This study should benefit those who are concerned with development and adaptation of method support technologies.

1 Introduction

An increasing awareness of the benefits of process support in metaCASE environments has taken place in recent years. MetaCASE environments are systems for method specification and support, but until recently their view of methods has been overwhelmingly product-centred (Koskinen and Marttiin, 2000). The emerged studies cover a wide range of scattered, miscellaneous issues of interest. These include fine-grained processes (Froelich et al., 1995), modelling guidance (Rolland et al., 1995; Si-Said et al., 1996; Pohl, 1996; Pohl et al., 2000), traceability (Jarke et al., 1994), metrics (Laamanen, 1995), reusability (Rolland and Prakash, 1993; Rolland et al., 1998), task-level design processes (Wijers, 1991; Marttiin, 1994), process maturity (Skelton, 1995; Kumar, 1995), metamodelling (Mi and Scacchi, 1996; Koskinen and Marttiin, 1997), method assembly (Harmsen et al., 1994; McLeod, 1995; Brinkkemper et al., 1999),

environment integration (Jarke and Rose, 1992), and simulation and enactment (Scacchi, 1996).

Most of these issues and perspectives are directly derived from existing software process research. This is of course an obvious choice, but tends to obscure some metaCASE specific concerns. The key concern in metaCASE is adaptability of methods alongside the natural evolution of local development approaches and practices. The need for customisation and evolution of process models is widely recognised in process research. It is a central theme in many process support environments (Bandinelli et al., 1993; Conradi and Jaccheri, 1993; Finkelstein et al., 1994; Kaiser and Ben-Shaul, 1993). Some concerns relate also to the customisation of metaprocesses, i.e., the processes of process engineering itself (Lonchamp, 1995). However, this is not the case with customisation of languages and techniques for process modelling. Varying modelling needs are expected to be met either by introducing a set of divergent process modelling languages from which to choose. There are also some general language designs that contribute to linguistic coverage (Sutton et al., 1995; Conradi and Liu, 1995).

In a series of empirical studies, Jaccheri et al. (2000) find that software development organisations have defined processes, and they use local languages and sometimes tools to represent their processes. However, the organisations are not using software process technology to support their process, although their potential benefits are admitted. In another empirical study, Phalp and Shepperd (1994) find it beneficial to take into account the characteristics and needs of the organisation when choosing a process modelling approach. This allows one to identify appropriate notations and modelling strategies. Rossi and Sillander (1998b) find it necessary to engineer a process modelling language in accordance to the process context. This abides to the conclusion by Sharp et al. (1999) that an effort aimed at improving software development processes needs, to be successful, to recognise the cultural context and to make explicit the software practices as they are actually understood and applied by software developers. Comparable empirical findings have been made of the use of method support in metaCASE (Smolander et al., 1990).

Lack of input from general IS research and from sociological and psychological research manifests itself in current process research. Process quality is reduced into an instrumental consideration of the productive capabilities of an organisation, regardless whether it is viewed from a technical (products and processes), linguistic (communication), or organisational (interplay of organisational agents) viewpoint. Social quality that deals with the motivational capabilities of an organisation is widely ignored in quality improvement. Human issues have become a serious concern in process engineering only recently (Sharp et al., 1999; Derniame & Kaba, 1999). Consequently, the diversity and variability of 'meta-information systems', and their impact on the feasibility of process approaches, are not recognised well enough.

The justification for language adaptation is derived from organisational and social considerations, that is, on the role of technology and process improvement in larger contexts. Current process technologies tend to be rigid

in the process thinking they promote, and hence to become an obstacle in organisational improvement and evolution. Thus, extensive customisability should be emphasised especially in metaCASE. In this article we discuss a generic process modelling and enactment system. Fully operational, such system would increase the ability to respond to local adaptation needs even if the required modifications concerned the process modelling approach itself.

The purpose of this article is to present a prototypical generic process modelling and enactment system designed for a metaCASE environment, and to assess the prototype against a domain framework for customisable method support environments. The domain framework collects a set of criteria to assess such method support environments.

This study takes a constructive research approach. The research process is illustrated in figure 1. The research started with an initial theory building phase that involved literature reviews of process modelling languages (Marttiin, 1994b; Koskinen, 1996a, 1996b). It was followed by prototype development that was conducted in two phases of design, implementation, and testing (Koskinen and Marttiin, 1997; Marttiin, 1998b). Each testing stage contributed to further theory building, and some design experiments were made.

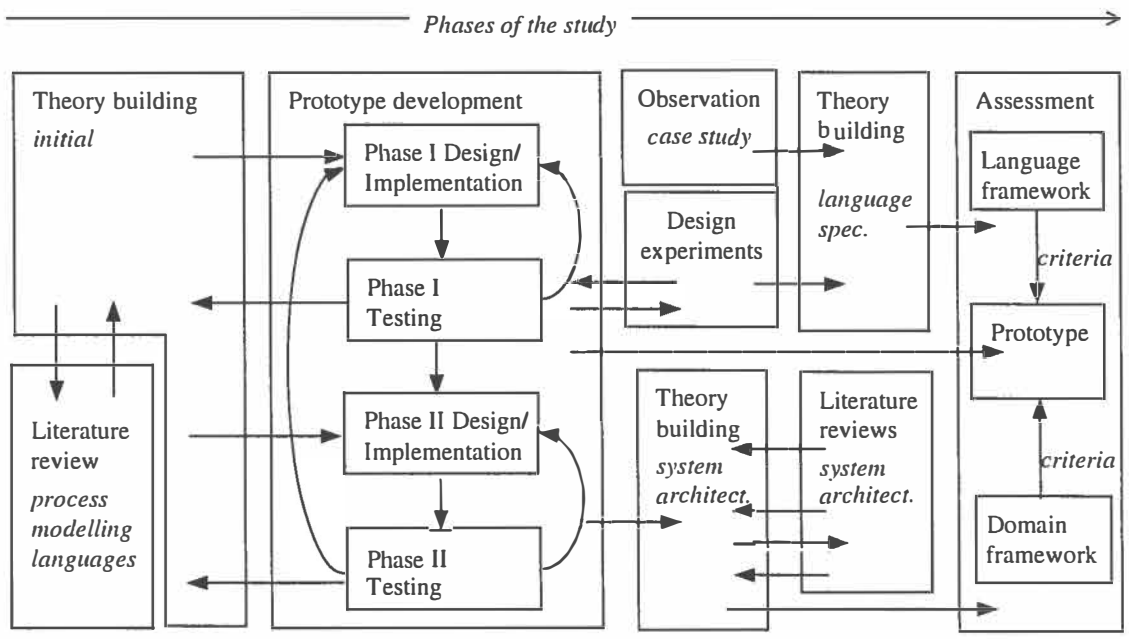

FIGURE 1 The research process of the study

When the prototype had reached an adequate level of robustness it was used for more comprehensive design experiments. Simultaneously, we followed a related case study conducted in a software development organisation (Rossi and Sillander, 1998a; Rossi and Sillander, 1998b). The design experiments and observations contributed to further theory building on language specification (Koskinen, 2000b). The theory building phase resulted in a generic language model (Koskinen, 2000a).

Meanwhile, the experience gained from developing the prototype led to theory building concerning relevant system architectures (Marttiin, 1997; Koskinen and Marttiin, 1998). Three literature reviews were conducted that contributed to theory building and the development of a general architecture for a customisable design environment (Marttiin, 1997; Koskinen, 1999; Koskinen and Marttiin, 2000c). The language model and the general architecture formed the basis for a domain framework. This framework contains a set of assessment criteria for customisable method support environments. Finally, the prototype was assessed against these criteria to reveal areas for further development.
This paper reports the latest phase of the research process. First, we discuss the metaCASE environment, MetaEdit+, and the prototypical process modelling and enactment system, CPME (Section 2). Thereafter, we introduce a domain framework that collects criteria for assessing customisable method support environments (Section 3). Thereafter, we assess CPME and MetaEdit+ against the domain framework (Section 4). Finally, we examine the results to identify a body of central themes for future research in the area (Section 5).

2 A Generic Process Modelling and Enactment System for a MetaCASE Environment

In this section we illustrate a generic process modelling and enactment system called Customisable Process Modelling Environment (CPME). It is a prototypical process support system designed for a metaCASE environment, MetaEdit+. First, we shortly describe MetaEdit+ to give an overview of the system, with which CPME is intended to operate (Section 2.1). Thereafter, we briefly illustrate the core parts of the CPME system (Section 2.2). We begin our detailed discussion with the process meta-metamodel of CPME: the GOPRR-p model (Section 2.3). We describe GOPRR-p metatypes, discuss the reasons that lead us to reuse the design of the GOPRR model, and some drawbacks entailed. We also make some suggestions for further improvement of the GOPRR-p model. Thereafter, we illustrate the process modelling and enactment system in detail (Section 2.4). We describe the use of notational constructs in process modelling, and the use of semantic constructs in process enactment. We then make some suggestions for their further improvement.

2.1 Overview of MetaEdit+

MetaEdit+ (Kelly et al., 1996) is a commercial multi-user metaCASE environment[1] that contains support for method engineering (CAME) and systems modelling (CASE). It is based on a client-server architecture. Each client contains a set of tools and a MetaEngine. MetaEdit+ tools handle

[1] MetaEdit+ is a registered trademark of MetaCase Consulting Ltd.

representational data but request all operations on conceptual data from the MetaEngine. Clients do not communicate with each other directly but through the shared design information. The repository in MetaEdit+ is implemented as an object-oriented database running at a server (Kelly, 1998). Both design models (instances) and design methods (types) are stored in a repository. Modification of any design information in one MetaEdit+ client is automatically reflected to other clients on the commitment of a transaction. Repository maintenance is supported by a set of tools that provide the necessary operations for browsing models and their components, deleting models and representations, and executing repository transactions.

CAME Functionality. The meta-metamodel underlying MetaEdit+ is the GOPRR model. It is based on a simple conceptual foundation of a few metatypes for defining methods. The model includes five metatypes: Graph, Object, Property, Role, and Relationship[2]. A graph type is a collective primitive that specifies a technique. Object, role, and relationship types specify method elements in a technique and property types are used to describe these elements.

Methods in MetaEdit+ are integrated collections of modelling techniques. The method elements are collected in a graph type and rules are given for their combination in graphs. Hierarchies of models (decomposition) and connections between models (explosion) are specified by creating link types from object types into graph types, and by attributing property types to object types. The GOPRR-p makes a distinction between conceptual and representational method knowledge. Each method component can involve several representation styles: graphical, matrix, and tabular (Marttiin et al., 1995).

The CAME tool set in MetaEdit+ includes form-based tools for creating and managing method components and composing them into method specifications (Rossi, 1998; Zhang, 2000). The tools are based on the GOPRR model and methods are defined using the GOPRR metamodelling language (Kelly, 1998). The heart of the metaCASE architecture is MetaEngine that embodies a GOPRR implementation. The MetaEngine provides interface services for the tools, e.g., selection dialogs and warnings according to the method specification.

CASE Functionality. Basic CASE tools in MetaEdit+ include editors for system modelling (Diagram Editor for graphics, Matrix Editor for matrices, and Table Editor for tables), and tools for browsing (Repository Browser), reporting (Report Editor), and querying (Query Editor) repository data. In addition, MetaEdit+ contains a hypertext subsystem for model linking and annotation (Oinas-Kukkonen, 1997; Kaipala, 2000).

Production support in MetaEdit+ includes representation, analysis and transformation functions. First, the representation function consists of operations that focus on constructing system models. It covers both model level (graphs) and model element level (objects, roles, relationships, and properties) operations. The operations deal both with conceptual and representational data. MetaEdit+ maintains several representational versions of a conceptual

---

[2] In the following, instances of types are in lowercase, e.g. *object*, types are appended with 'type', e.g. *object type*, and metatypes are initially capitalised, e.g. *Object*.

model, where each diagram, matrix and table may contain a subset of the model concepts. Versioning of conceptual models is not yet supported.

Secondly, analysis function consists of operations that concern ensuring the correctness of system models. Analysis is either automated or manual. MetaEngine manages automated support by enforcing GOPRR rules (meta-metamodel) and method rules (metamodels). Manual analysis is carried out using reporting functionality. The GOPRR model determines the general rules of how metatypes are mapped together. Method specific rules ensure that system models will be created consistently. The consistency rules are checked when creating and updating model elements. MetaEngine provides modelling guidance such as selection dialogs and warnings and prevents the user from performing illegal operations.

Thirdly, the transformation function deals with operations that aid in generating textual specifications based the system models, such as code and document generation. Report templates are specified with the Report Editor that uses a GOPRR based reporting language.

Agent System. The agent system in MetaEdit+ is specified using agent models. Agent models are currently simple ones describing actors and their responsibilities in a project. They are based on the agent types *user*, *user right*, and *project* (Marttiin, 1994b; Marttiin, 1998b).

Basic operations on users (creation, removal, permissions) and projects are managed by the MetaEdit+ repository. A project is a root pointing to a set of graphs stored in a repository area (system models and/or process models created in a project). The only restriction for opening a project is that the user must be assigned to it. Users may have several projects open at the same time and they can move freely between the projects. Users are identified when they log into the system. A session consists of subsequent transactions each of which ends with committing or abandoning all changes made during that transaction.

MetaEdit+ supports asynchronous co-ordination. Its concurrency control is based on automatic locking strategies that use write locks. These are implemented at the model element level and are divided into conceptual and representational locks. Automatic locking, transactions and sessions are discussed in Luoma and Somppi (1996) and Kelly (1997).

2.2 CPME: Process Support System

The aim of CPME is to provide process-based guidance and support for production and co-ordination tasks in MetaEdit+. Its purpose is to facilitate understanding, guide users and support production activities and their dependencies (co-ordination) within a multi-user environment. It uses explicit (visible) process models and process metamodels, supports large-grained co-ordination activities, and enables incremental run-time customisation of process models and process modelling languages. We describe five core aspects of the CPME prototype.

Process metamodelling. Process metamodelling is the means to specify process modelling languages in CPME. The GOPRR-p model is the underlying

process meta-metamodel that constitutes the foundation for all process metamodels. Process metamodelling is facilitated by a set of form-based process metamodelling tools that are based on the GOPRR-p model They use a GOPRR-p based process metamodelling language. The tools are an extension to the Metamodelling Tools in MetaEdit+. This is a natural continuum entailed from a choice to reuse the GOPRR model in the design of a process meta-metamodel. The basis for this choice is discussed in Section 2.3.2.

Process modelling editor and enactment interface. CPME provides a generic process editor to facilitate flexible, incremental process modelling. It can be used both for process modelling and process enactment. A process metamodel specifies rules for both process modelling and process enactment. This has lead to the current decision to design and enact process models through the same interface. However, the design of GOPRR-p does not constrain the type of enactment interface. The actual interaction with the process engine is carried out through menus that could be attached to any tool in MetaEdit+ with minor effort.

Process engine and process enactment. The Process Engine in CPME implements the GOPRR-p model. It is generic in that it uses an explicit and separate process metamodel as a language definition. This is analogous to a generic interpreter that would use an explicit programming language definition to execute a piece of code written in that language. A process engine instance encapsulates process model components so that the engine is constructed and customised through constructing and customising a process model. In this way, no transformation is needed to make the process model enactable. Also the effort needed in specifying the process engine instance is reduced.

Process programming interface. MetaEdit+ tools are evoked by the Process Engine through the process programming interface of a tool. This interface specifies all operations that a tool can provide for process support. The tools include all CASE and CAME tools as well as CPME tools. When tools are integrated to CPME, appropriate operational interfaces must be coded into the tools. However, the process programming interface is intended to be customisable but the customisation mechanisms are not yet implemented.

Enhanced agent system. CPME enhances the basic user model of MetaEdit+ with user roles. Users may participate in a project in several roles. A user role describes a position and function in a project and is used in controlling participation in a process by assigning tasks to user roles. The project model is extended with two lists: one for mapping users to their roles, and another for mapping user roles to assignments. Agent models are managed through form-based tools: *User Role Tool, Task Assignment Tool,* and *Project Tool.*

Agent types could be extended with additional properties for project management's needs (e.g., users' experience histories, costs of work hours). However, in their current form they provide the control needed in process models, i.e., assigning tasks to specific user roles.

2.3 GOPRR-p: The Process Meta-Metamodel

CPME uses a process metamodelling approach for the customisation of process modelling languages. Earlier, metamodelling approaches have been used for the customisation of conceptual frameworks and notations for data modelling (Sorenson et al., 1988; Kelly et al., 1996; Nissen et al., 1996). In CPME, process metamodels cover the conceptual framework, the notation and the semantics of a process modelling language (Koskinen, 2000a). A conceptual process metamodel captures the representation independent rules of model composition. A notational process metamodel captures the modelling notation in a specific process modelling technique. A semantic process metamodel captures the rules of model interpretation and enactment. Thus it can be seen as an aid or a mediator that assists a generic process engine with the interpretation of process models.

The GOPRR-p model distinguishes between user related and environment related concepts in a conceptual process metamodel (Koskinen and Marttiin, 1997). User-related processes are performed by a human and environment related processes are performed by an automated environment. Despite the differences between the requirements of human and machine interpretation, a process model should fully support both processes during model enactment. CPME distinguishes user-focused and environment-focused process models and languages. The former are used to define human processes and the latter to define automated processes. Since these models and languages are conceptually overlapping, we have decided to specify the overlapping parts within the scope of the former to avoid redundancy.

TABLE 1 User-focused and environment-focused languages are covered by different components of the GOPRR-p model.

<table>
<tr><th>Notational (user)</th><th>Conceptual (user)</th><th>Conceptual (environment)</th><th>Semantic (environment)</th></tr>
<tr><td>Diagram</td><td colspan="2">Process Graph</td><td>Manager Pattern</td></tr>
<tr><td rowspan="2">Symbol</td><td rowspan="3">Process Element</td><td></td><td>Progress Pattern Feature</td></tr>
<tr><td>Action</td><td>Function Pattern</td></tr>
<tr><td>Colour</td><td>State Model</td><td>State Pattern</td></tr>
<tr><td>Symbol</td><td colspan="2">Relationship</td><td>Progress Pattern Feature</td></tr>
<tr><td>Symbol + Line</td><td colspan="2">Role</td><td>Progress Pattern Feature</td></tr>
<tr><td rowspan="2">Label</td><td rowspan="2">Property</td><td></td><td>Value Pattern</td></tr>
<tr><td>Action</td><td>Function Pattern</td></tr>
</table>

Table 1 shows how the two types of processes are covered in the GOPPR-p model. A user-focused language is covered by notational and conceptual process metamodels, whereas an environment-focused language is covered by conceptual and semantic process metamodels. A conceptual process metamodel

thus integrates a user-focused language and an environment-focused language. It also constitutes the core to which notational and semantic process metamodels are attached. All process metamodels constructed in CPME are based on the GOPRR-p model.

2.3.1 GOPRR-p Metatypes

The process meta-metamodel GOPRR-p incorporates conceptual, notational, and semantic metatypes. Each class of metatypes concerns specific information of process models: their composition, visual appearance, or interpretation and enactment. First, the conceptual metatypes are the core of the GOPRR-p model to which notational and semantic metatypes are attached. The current version of the GOPRR-p model supports six conceptual metatypes: *Process Graph, Process Element, Action, State Model, Relationship, Role,* and *Property*. Second, CPME provides three representation styles: diagram, matrix, and table. Notational constructs in each style are generically classified as views, view fragments, and visual attributes. The core style is diagram and it is used in the Process Editor. The notational metatypes in diagram style are *Diagram* (view), *Symbol, Line* and *Label* (view fragments), and *Colour* (visual attribute). Many of the notational rules governing the dependencies between notational constructs are given in the GOPRR-p implementation. Third, semantic construct types are patterns of interpretation and thus the semantic metatypes in the GOPRR-p model are a set of pattern types: *Manager Pattern, Progress Pattern, Function Pattern, State Pattern,* and *Value Pattern*. Together, semantic construct types specify a unique enactment paradigm that comprises the rules and constraints governing model interpretation and enactment.

In the following, we discuss each conceptual metatype together with the related notational and semantic metatypes.

Process Graph. A process graph is a model of some restricted part of a process: a process model is composed of a set of graphs. A graph type integrates the components of a process metamodel. The graph structure (i.e., dependencies between process elements) is accomplished by a binding mechanism: every relationship involves a set of roles, which further are participated in a set of process elements. The notational metatype related to Process Graph is Diagram. Diagrams are views that show the arrangement and relations of different symbols, lines, and labels. The semantic metatype related to Process Graph is Manager Pattern. A manager pattern is a construct that controls the interplay of several successive or simultaneous progress patterns. The enactment of a process graph is thus determined by its manager pattern and the progress patterns of its components.

Process Element. A process element specifies any component of a *user* process, such as a task or a deliverable. The notational metatype related to Process Element is Symbol. Symbols are independent graphical 'nodes' in a diagram. The semantic metatype that is attached to Process Element is Progress Pattern. A progress pattern is a construct that specifies the advancement of an execution thread at a specific point along an execution channel. In CPME,

progress patterns compose a set of features. Each feature specifies a small generic aspect of enactment: either of advancement or co-ordination of concurrent execution threads. The progress pattern for process elements combines a set of advancement and co-ordination features.

Action. Actions are attached to a process element to specify how an automated environment is intended to support the process element. A process element type specifies an integration constraint that determines, to which kind of actions it can be related. It enables flexible integration that need not be wholly determined on the level of language. Actions are used to evoke different kinds of tools to operate on some products. Actions are represented in separate form views. The semantic metatype that is attached to Action is Function Pattern. It specifies how a tool operation is evoked on a certain set of products. A function pattern specifies the type of operation invocation and the mode of control during operation execution. In the current implementation, all function patterns are envelopes for black-box operations and hence only low level of control can be provided.

State Model. A process element type specifies a state model that determines the possible states and state transitions during a process element's enactment life-cycle. The notational metatype related to State Model is Colour. The colour of a process element symbol is determined according to the enactment state of the corresponding process element. The semantic metatype that is attached to State Model is State Pattern. A state pattern specifies how the enactment state of a process element of certain type is changed in response to internal events.

Relationship. A dependency between concepts within a conceptual framework specifies how concepts relate and affect each other. The GOPRR-p model manages dependencies through bindings as mentioned above. A relationship represents a dependency between a set of process elements. It is represented using a symbol. The notational metatype related to Relationship is Symbol, whereas the semantic metatype is Progress Pattern. A progress pattern for a relationship type combines a set of advancement features. These features are unique for the execution of relationships.

Role. A role connects a process element to a relationship and defines how the process element plays part in the relationship. The notational metatypes related to Role are Symbol and Line. Lines are 'edges' between symbols. The semantic metatype that concerns Role is Progress Pattern. A progress pattern for a role type combines a set of advancement features unique for the execution of roles.

Property. Properties form a means to store process specific data, to refer to different kinds of data and objects, or to store collections of data and objects. Properties may appear graphically as textual labels such as names, text fields or numbers. The notational metatype related to Property is Label. The semantic metatype that is attached to Property is Value Pattern. A value pattern specifies the formation and the role of property values in process enactment. Action types can be attached to property types to specify property constraints and

value calculation. In contrast to actions with process elements, the integration is fully specified at type level.

A detailed discussion of process metatypes is found in Koskinen (2000).

2.3.2 Why GOPRR?

The design of the GOPRR-p model is based on the GOPRR model. There are several reasons for this. Firstly, an examination of different views of processes and process modelling languages gave some directions for the representation capability of the required model. Much of the fixedness in existing languages was found to result from the use of binary dependencies that are capable to create a binding only between two process elements. Therefore, information about n-ary dependencies (such as needed for *branch* and *merge*), and thus also about the binary dependencies of which the n-ary dependencies are built, have to be specified as part of process elements. GOPRR's view on dependencies is very elaborate and it allows any n-ary dependency to be specified independently of objects. Furthermore, it distinguishes between relationships and roles and specifies all relationships and roles as autonomous entities. Therefore, the binding structure in GOPRR is highly flexible. Most available models were weak in this regard.

Second, process metamodelling has to be capable of addressing the notation, conceptual framework, and semantics of languages. It should also enable flexible reuse and combination of process metamodel components. Moreover, representation independence has to be supported. GOPRR differentiates between the notation (for representations) and the conceptual framework (for conceptual models). Even though the implementation of GOPRR does not fully support this capability, it is considered and taken into account in the design of GOPRR. Other available models did not support the distinction between representations and conceptual models, nor did they allow representation independence.

Third, a process meta-metamodel should neither be too generic nor too strictly bound to a specific ontology. A generic model (such as O-Telos in Jeusfeld, 1993) can be used to define very elaborate metamodels, but it also makes metamodels very complex to understand, manage and evolve. An ontologically bound model, on the other hand, restricts its applicability to a narrow perspective of processes (such as viewing a process as a network of activities). The GOPRR model was an appropriate alternative since it is neither as generic as in O-Telos, nor as ontology-bound as the generic models used in existing process support systems.

Fourth, the actual application of models was considered. Many metaCASE developers have had problems in transferring their technology into real use. Also, the metamodelling process appears to be quite time-consuming. In contrast, MetaEdit+ is a commercial metaCASE environment that has currently several hundred users in more than 30 countries around the world. Therein, GOPRR has shown its strength as implementing over 30 methods, each including one or more modelling techniques. GOPRR's ancestor, OPRR

(Smolander, 1992), has been used to implement even more modelling techniques. The time needed for metamodelling is very short in MetaEdit+ in comparison to other metamodelling tools. For example, even complex methods, such as Unified Modelling Language (Booch et al., 1999) can be developed by a skilled method engineer within a few days. Therefore, GOPRR's capability to represent a wide variety of modelling techniques and to support rapid method engineering is ensured.

Fifth, the design of the GOPRR model was found to follow such principles that changes and extensions are fairly easy to make both in system design and implementation. This ensured that future improvement of the model is possible. A process meta-metamodel could be based on the GOPRR model without unnecessarily restricting later improvements. That was a core concern in the iterative and cyclical prototype development. Also, reuse of design and implementation components gave benefits (such as compatibility and consistency) when the approach was implemented. However, this is considered only as an additional advantage – not a reason – for choosing GOPRR as the basis for a process meta-metamodel.

2.3.3 Drawbacks of the Choice to Reuse the GOPRR model

Nevertheless, the choice of reuse has been not only an advantage. There are always some design decisions that are not the best for prototyping. It is not easy to get approval for changes that would benefit prototype development, but that would require rework on a commercial system. The changes needed by the prototype must therefore be made in a research version of the system. Consequently, when the commercial system is updated and the changes propagated to the research system, there is always extra rework in ensuring that the changes do not override changes made for the prototype. To reduce this rework to the minimum, compromises are necessary.

Also, the choice of reuse restricts the feasible starting point for prototype development. Some such issues are discussed by Marttiin (1998a). Firstly, the model chosen determines the level of support that an approach provides. Increasing the level of support requires a richer and more complex process metamodelling language and a more detailed and laborious metamodelling process. Therefore, it is necessary to examine what is the level of support that a GOPRR based model can provide for process support and whether there is need for further modification.

Secondly, it necessary to distinguish between the restrictions imposed by GOPRR's design and GOPRR's implementation in MetaEdit+. The current design of GOPRR is much more elaborate than its implementation. This concerns support for notational customisation and conceptual rules. The problems faced due to the reuse of GOPRR concern more GOPRR's implementation in MetaEdit+ than GOPRR's actual design. For example, GOPRR's bindings in the implementation are structures that keep objects together but do not allow direct communication between objects. Also, GOPRR's graph as implemented is a collection of objects that do not directly

know in which graph(s) they appear. These implementation decisions have been made to allow more flexible reuse, but they also inevitably hamper fluent enactment. However, the design of GOPRR does not impose these restrictions.

When we assess the amount of time and effort needed for adapting GOPRR in comparison to some other model, we conclude that the required modifications are less laborious in the case of GOPRR. As the main objective has been to find a starting point for several cycles of elaboration and evolution, instead of a ready-made model, we regard the decision to reuse GOPRR well-founded.

2.3.4 Suggestions for Further Improvement of the GOPRR-p Model

We find several ways in which the GOPRR-p model should be improved or extended regarding its current implementation in CPME prototype. In the following, the suggestions for improvement in the current version of the model are discussed for each metatype class individually.

Conceptual metatypes. The suggestions for improving the conceptual part of the GOPRR-p model concern mainly the coverage of possible concepts and the level of customisability. Firstly, the GOPRR-p model does not cover a distinction between a process and its context. It should distinguish between concepts that form the core of a process model, and concepts that enable a "connection" to the context that is not expressed in the model. Due to the lack of "external concepts", certain external dependencies (such as detecting events in the external system) cannot be adequately specified.

Secondly, we find that in certain cases a complex element would be a more suitable construct than a pair of a process element and a process graph. Partly, this is a notational problem: a symbol representing a process element cannot embed symbols of process elements in its decomposition in a process graph. However, there are cases in which the use of a process graph as the decomposition of a process element is not fully justified. The separation of a process element and a process graph should always be based on a conceptual distinction between a process element and its decomposition (as in the case of an interface and its possible implementation). If a concept is "produced" by its decomposition, separation is not conceptually justified.

Thirdly, state models should be more elaborate. In the current implementation there is a fixed base-model that can be modified only by replacing the names for states. For example, if we are familiar with the state model presented by Mi and Scacchi (1992), we can rename *Available* (in the base-model) to *Ready*, *Finished* to be *Done* and *Exception* to contain two state types *Stopped* and *Broken*. However, the transitions specified by the base-model cannot be changed. In the future, the system should support specification of various state-transition models for process elements.

Fourthly, GOPRR-p supports the specification of property types for properties that compose a list or collection of properties of certain type, but not property types for properties that compose different types of properties. In

other words, it is not possible to define a property type that would combine, for example, start date, end date, and duration calculated from the first two, to be reused between different process element types.

Notational metatypes. Up to date, the notational part of the GOPRR-p model has fully relied on the design of GOPRR, while we have concentrated on more critical aspects of modelling: composition and semantics. Therefore the following suggestions are not as detailed as those concerning conceptual and semantic metatypes. However, we have encountered several inconveniences during our experiments on process metamodelling and process modelling.

Firstly, the notational process metamodel should be separated from the conceptual one. Currently, there are fixed mappings between notational and conceptual metatypes. For example, the GOPRR-p model enforces that all process elements are represented as symbols and all bindings basically as lines. This is a problem since there are many notations that represent process elements as lines, not symbols.

Secondly, in many aspects we have needed to adapt notational rules when they are implemented in a process metamodel. This is due to notational rules that are currently fixed in the GOPRR-p implementation. Customisability of those notational rules should be enabled. Also, we have found several notational rules that currently cannot be specified. There is a need for new types of rules, such as those governing the location of view fragments in relation to others. Especially, customisability of notational rules concerning dependencies between notational constructs should be improved.

Thirdly, there is a need for visual dependencies, especially a group mechanism that would enable the creation of composite notational constructs within a diagram.

Fourthly, we suggest that the customisation system for notations is generally made more flexible. Different types of view fragments could be combined in customisable representation styles. The system could also have mechanisms that enable the definition of new types of views and view fragments. This would allow more flexible representation styles.

Semantic metatypes. The suggestions for improving the semantic part of the GOPRR-p model, mostly concern inadequate refinements.

First, we have found some misjudgements that are due to handling features originally as a single class. Co-ordination features, such as the one specifying initiating process element types, at first sight seem to concern type-specific information. However, the feature concerns a specific process element type *only in the context of* a certain process graph type. When co-ordination features are related to process elements instead of process graphs, the reuse of process element types is difficult. A process element type that is otherwise perfectly reusable cannot be reused in another process graph type, if it is fixed as initiating while the reused one should not be, and vice versa. Co-ordination features should be attached only to manager patterns and advancement features to progress patterns.

Second, the feature sets for each conceptual metatype are currently fixed. The GOPRR-p model could address a wider range of languages if customisation of the feature sets were allowed. Also, the system could benefit from allowing new features to be specified.

Third, there is a lack of semantic metatypes for mediating information across environment borders. The system would benefit if the GOPRR-p model allowed one to specify patterns for event detection and data exchange. Function patterns can currently evoke only MetaEdit+/CPME tools and handle products produced by those tools.

Fourth, the improvement of state models requires also respective improvement of state patterns.

Fifth, different types of function patterns should be introduced in the GOPRR-p model. There should be enveloped functions for 'black-box' execution of tools and controlled functions for 'white-box' execution. It would also be beneficial to introduce controlled functions specifically designed for reflexive operations. The current implementation supports only controlled functions. The functions can evoke reflexive operations only through invoking the Process Editor, not referencing the process model (in which the action instance locates) directly.

Sixth, the addition of complex properties requires a suitable pattern for managing complex values. Currently, we have to use products (GOPRR objects) to specify complex structures without redundant enactment mechanisms. This is not a sound solution practically, nor theoretically. Furthermore, we have found that it would be feasible to have specific logic functions for performing logical operations on complex values.

Last, value patterns might be distinguished into active and passive value patterns. Currently, all value patterns are basically active. If there is no function defined for the value pattern, the function mechanism is simply not used. However, we find this conceptually not justified. Although we are not fully convinced of its necessity, a weakness in conceptual distinction is always an open invitation for problems during later extension and improvement.

2.3.5 Process Metamodel Abstractions

A major hindrance in process metamodelling in the current CPME prototype is narrow support for metamodel abstractions. Basically, all that is supported is sub-typing. We have found that this is not adequate for some process modelling languages. Our experiments have shown that there is a need at least for fragment patterns, "context inheritance", and type grouping. First, a fragment pattern specifies a metamodel for a specific type of graph fragment. The rules of a fragment pattern apply only to such fragments and they overrule all metamodel rules generally specified in the process metamodel. Secondly, context inheritance denotes that a specific type "inherits" its dependency context (the dependency types in which it may engage) from one context to another, but not necessarily to all contexts in which the type appears. In other words, there is a set of dependency types in which the type may engage in all

dependency contexts, but also an individual set of dependency types for each possible sub-context. Thirdly, type grouping denotes that a set of different types have a common dependency context. All these would reduce redundancy in process metamodels. Sometimes, a fragment pattern is also useful to avoid recursion in metamodels.

We also find it could be useful to introduce metamodel patterns to specify a set of generic metamodel rules for a common set of process metamodels. This would enhance metamodel reusability.

2.4 Process Modelling and Enactment System

The core of the generic process modelling and enactment system in CPME consists of the Process Editor and the Process Engine. The process editor is a combined editor and enactment interface. The Process Engine is a combination of a metaengine and a generic process interpreter. The relation between a metamodelling system and a process modelling and enactment system is shown in figure 2.

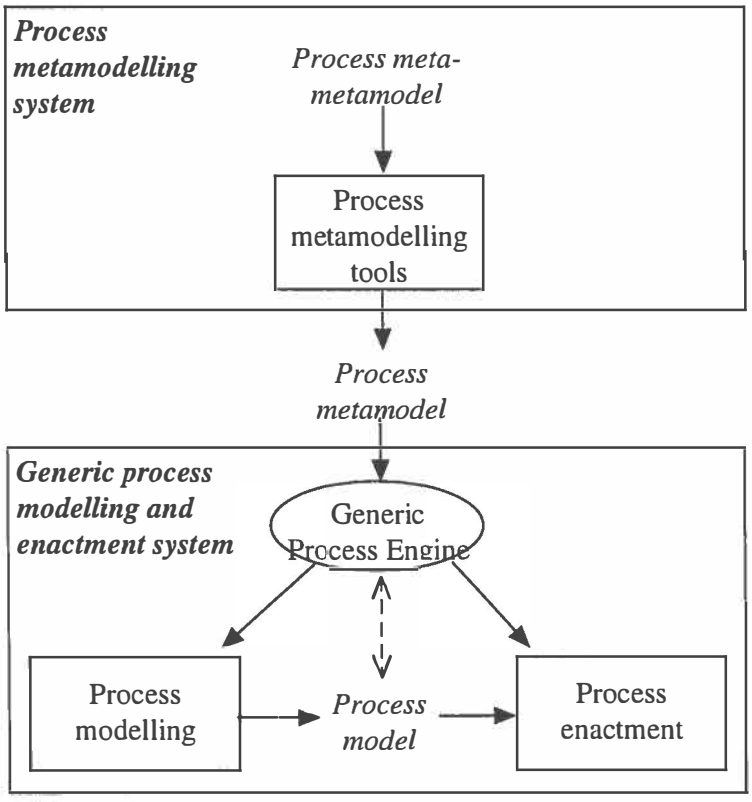

FIGURE 2 The relation between a process metamodelling system and a generic process modelling and enactment system.

A process metamodelling system supplies the generic process modelling and enactment system with a process metamodel. The process metamodel specifies both rules for process modelling and for process enactment. The process

metamodel configures the generic process engine that guides both process modelling and process enactment accordingly. As a metaengine, the Process Engine controls the creation and modification of process models, and as a process interpreter it controls the execution of process models.

A process engine instance encapsulates the components of a process model. The process engine instance is constructed and customised through constructing and customising the process model. This design choice entails that no transformation is needed to make process models enactable. It also reduces the amount of effort needed in customising a process engine. This integrated design has also lead to our decision to support the design and enactment of process models through the same interface, the Process Editor. However, the design of GOPRR-p does not restrict the kind of enactment interface to use. Interaction with the process engine is carried out through menus that could be attached to any tool in MetaEdit+ with minor effort.

2.4.1 Use of Notational Constructs in Process Modelling

The Process Editor is a customisable tool for process modelling. It implements a generic tool architecture that is configured by supplying it with a process metamodel. The core of the tool architecture is an interface layer that implements the generic process pattern of the GOPRR-p model. As a process representation is opened in the Process Editor, it is "hooked" to the components of the interface layer. Opening a process model in the editor involves that the corresponding process metamodel is automatically loaded into the editor. The Process Editor is literally generic: it cannot function without a process model and a process metamodel.

A process model may have several representations that give either a full or a partial view to it. A representation consists of a view and several view fragments. View fragments can be characterised by visual attributes. We discuss here the diagram representation style that is used in the Process Editor.

Diagram. A diagram is a collection of symbols, lines, and labels that are partially connected through various notational dependencies.

Symbol. A symbol may store a location point. The location of a symbol can change in a diagram. A process element symbol always stores a location point and thus it can be moved only manually. The location point of a relationship symbol is automatically optimised unless the location is chosen manually by moving the symbol in a diagram. The location point of a role symbol is automatically calculated according to metamodel rules. The symbol is automatically rotated relative to the angle in which it connects to the process element symbol. All symbols may store a set of connection points that determine where an incoming line can be connected. In case no connection points are set, all lines are connected to the middle of the symbol.

Line. A line is specified as a series of touch points. The location of a touch point may be set manually. Otherwise, the location changes automatically when the symbols that it connects are moved. The end points of lines are not

stored since they can be automatically calculated based on the location and the connection points of the symbols they connect.

Label. A label is a textual symbol. The content of a label is either fixed in the metamodel, or it reflects the value of a certain property.

Colour. Colour is a visual attribute that can be attached to any symbol. However, in case it is attributed to a process element symbol, it can be automatically changed according to the enactment state stored in the corresponding element node.

All notational constructs are accessed and modified through the interface layer of the Process Editor.

2.4.2 Suggestions for Further Improvement of the Representation System

Most of the weaknesses of the representation system are entailed from weaknesses in the notational and conceptual parts of the GOPRR-p model, and are mostly due to a lack of emphasis in our research. Therefore it is also difficult to make detailed suggestions for their improvement.

The notational part of the GOPRR-p model and the mechanisms of the representation system should be designed more systematically in the future. This should be done according to the same design principles that we have used in designing other aspects of the system. Firstly, the design should introduce a set of generic and specialised mechanisms. Secondly, the design should characterise conceptual clarity and comprehensiveness, avoid redundancy, and look for conceptual justification for all discrimination and integration introduced in the design. Thirdly, all design iterations should increase the formality of the design. The research process should be conducted in a way that before starting a new iteration a throughout study is conducted to evaluate and collect criteria for further improvement of the design.

2.4.3 Use of Semantic Constructs in Process Enactment

All execution in CPME is based on messaging: different process engine components receive and forward signals that transmit information on the execution of the process model. For each type of semantic construct there is a corresponding enactment mechanism that is specially designed for the use of such constructs.

Manager mechanism. The manager mechanism encapsulates a process graph and uses a manager pattern to determine how to control the interplay of several successive and simultaneous process elements in the graph. The manager mechanism is illustrated in figure 3. Mechanism components are shown as boxes, while signals passing between the components are illustrated as arrows.

The core of the manager mechanism is an execution manager that co-ordinates the execution of a component hander, value handler, and a state evaluator. There are two kinds of input signals that an execution manager may receive: signals that request information on the enactment state, and signals

that request execution. Requests for state information are forwarded to a state evaluator. The graph state is deducted from the states of the component elements. Requests for execution are forwarded to a component handler. In case the request initiates the execution of the graph, a feature handler checks up what types of process elements are initialising in the graph type. The component handler then executes or enables the execution of all appropriate process elements. In case execution is already started, the component handler checks which components are currently enabled, and provides them as a collection to the requester. Also the collection of all elements in exception state can be requested. At all events, a value handler is invoked to check whether the event involves calculating some property value or checking some property constraint.

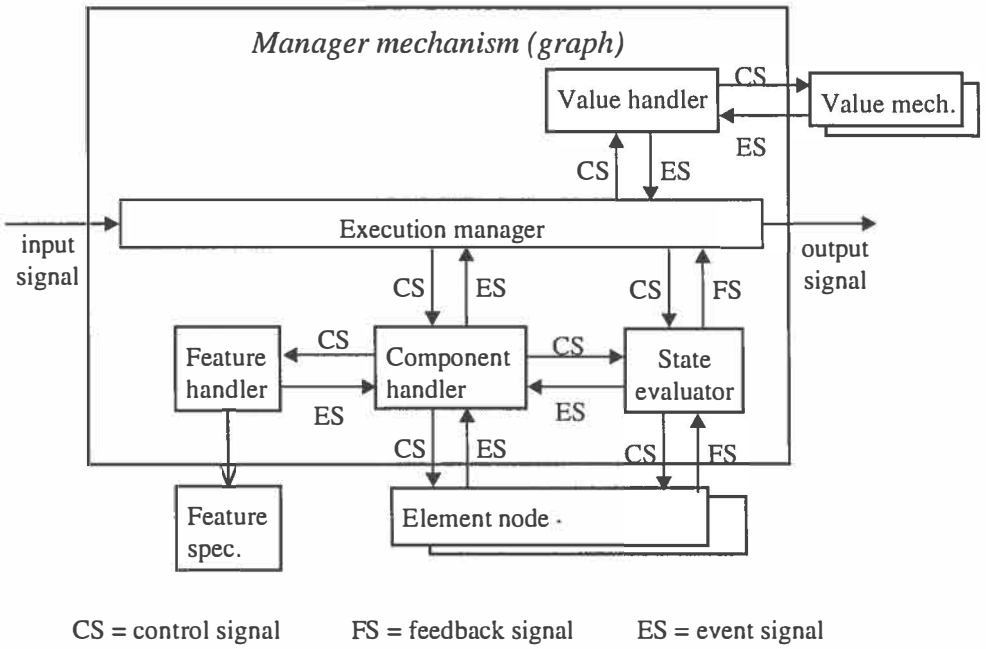

FIGURE 3 Manager mechanism component of the Process Engine

Element node mechanism. Each process element in a process graph is encapsulated into an element node that stores current state information of the process element. An element node can allow the initiation of several parallel execution threads. It collects all execution threads as they are initiated, and integrates them directly to the process model that is enacted. An element node stores its parallel sub-execution threads together with their individual enactment states. The use of sets for collecting parallel element nodes provides dynamism in process enactment. All parallel execution threads remain connected for later examination as process traces. The enactment node mechanism is illustrated in figure 4.

The execution manager of the element node mechanism co-ordinates the execution of a progress mechanism, an execution handler, a state mechanism, and a parallel handler. The progress mechanism and the state mechanism are

discussed later. The execution handler takes care of downward hierarchical execution channels. It aids in choosing alternative actions or decomposition graphs, and forwards execution thread to a function mechanism or a manager mechanism respectively. In case execution fails, the execution handler notifies the execution manager about it. The execution manager orders the state mechanism to change into exception state and suspends the enactment of the element node. The parallel handler is responsible for co-ordinating the execution and states of the parallel sub-nodes.

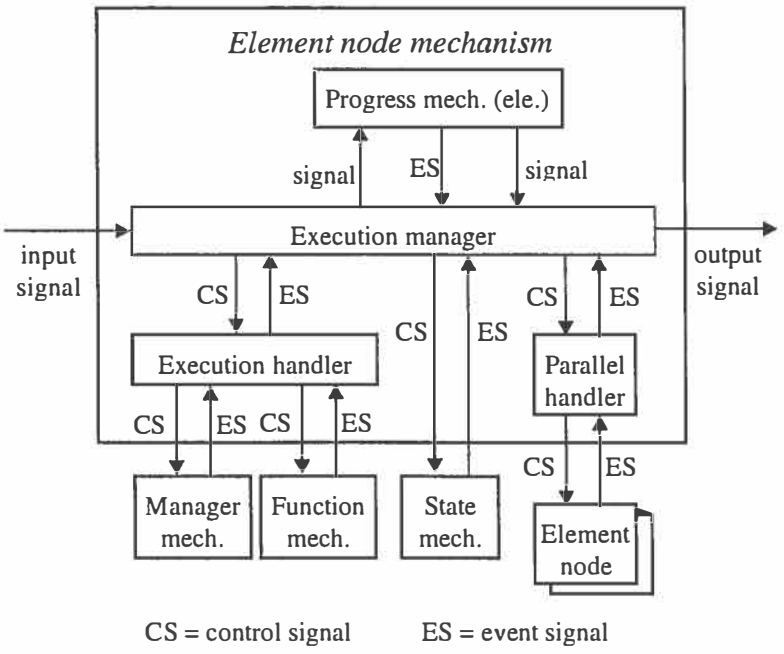

FIGURE 4 Element node component of the Process Engine

Enactment-time component reuse is supported by means of sharing and cloning. Process models can share element nodes with process elements or process elements only. In enactment sharing element nodes are shared. Thus all bindings and state information of a model component are shared between models. This allows co-ordination between different projects, processes, and perspectives. In element sharing process elements are shared by alternative models. In this case, enactment of the component in one model does not entail its enactment in another model, since it does not have the same bindings in each model. This allows specification of alternative process scenarios. Component cloning is used in the initiation of parallel execution threads. All the threads maintain their state information and the state of the modelled process element is composed from the states of the cloned ones.

Progress mechanism. A progress mechanism encapsulates either a process element, a relationship, or a role. It uses a progress pattern to determine how to advance an execution thread along a binding between process elements. A binding is created by connecting a set of process elements

to a set of roles and the roles to a relationship. This allows the use of n-ary bindings. The progress mechanism is illustrated in figure 5.

The execution manager of the progress mechanism co-ordinates the execution of a set of feature handlers and a value handler. A feature handler uses a feature specification to define how the signal is handled or modified as it passes through a specific feature point. The passing signal consists of information on the type of the thread (simple, parallel) and information for identifying which execution thread it concerns. The value handler functions in the same way as in the manager mechanism.

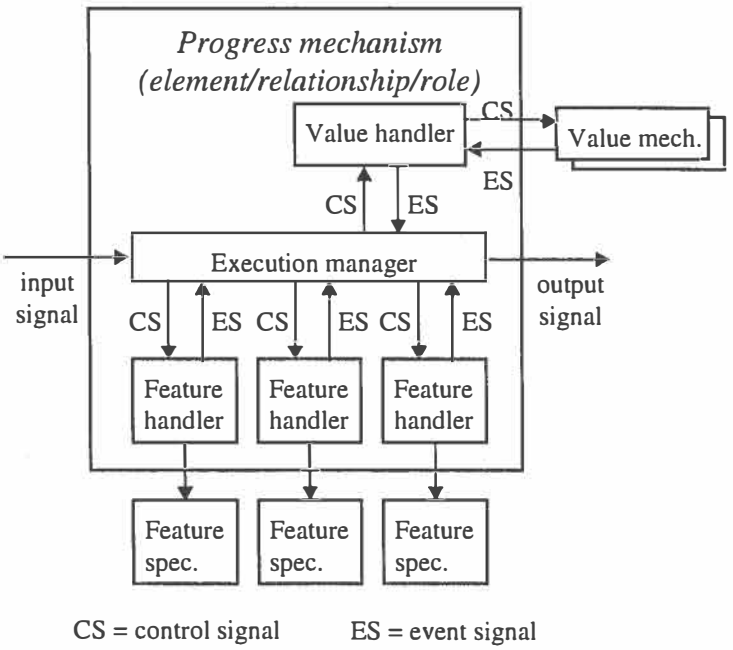

FIGURE 5 Progress mechanism component of the Process Engine

Function mechanism. The function mechanism manages invocation of tools. It uses a function pattern to determine the tool and the operation to invoke and the products that are passed to the operation as parameters. The function mechanism reconstructs the correct PPI call based on this information. The function mechanism is illustrated in figure 6.

The execution manager of the function mechanism manages a consistency controller and an execution controller. The consistency controller ensures that the tool specified in the function pattern is currently present in the system. If that is so, it negotiates with the tool to ensure that it still provides the requested operation and that the products are of correct type for the operation. The execution controller invokes the tool operation with the products, transmits information on the success of the operation, and receives the output products from the tool. The product handler takes care of all data transformations between the function mechanism and the tool.

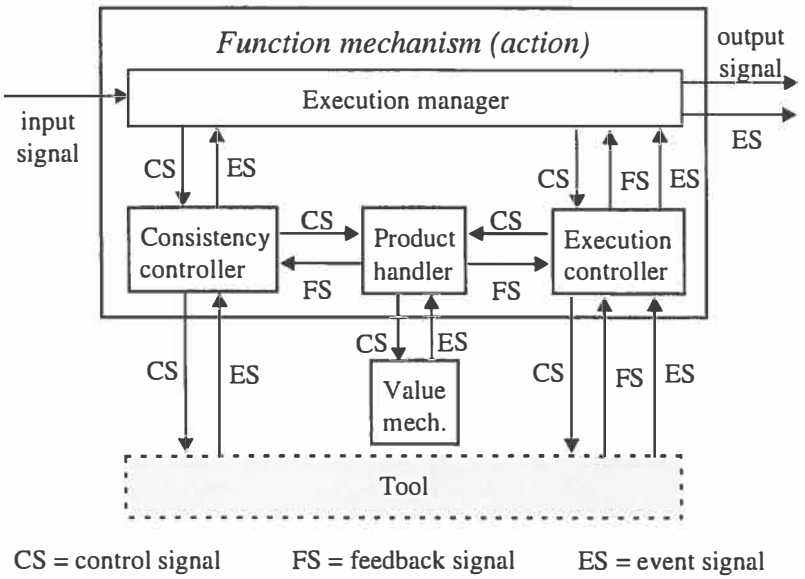

FIGURE 6 Function mechanism component of the Process Engine

Product sharing by identity is enabled by using a property as a product holder. An "empty" product holder denotes that some operation will concern a specific product but it does not yet exist or is not yet chosen. If the product exists and is known, a product holder can also be filled in advance. Process models can thus capture detailed information of the products. This facilitates customisation of enacting process models and manual state updates.

State mechanism. The state mechanism manages the transition of an element node's enactment state in response to internal events. It also handles all requests for information of an element node's state.

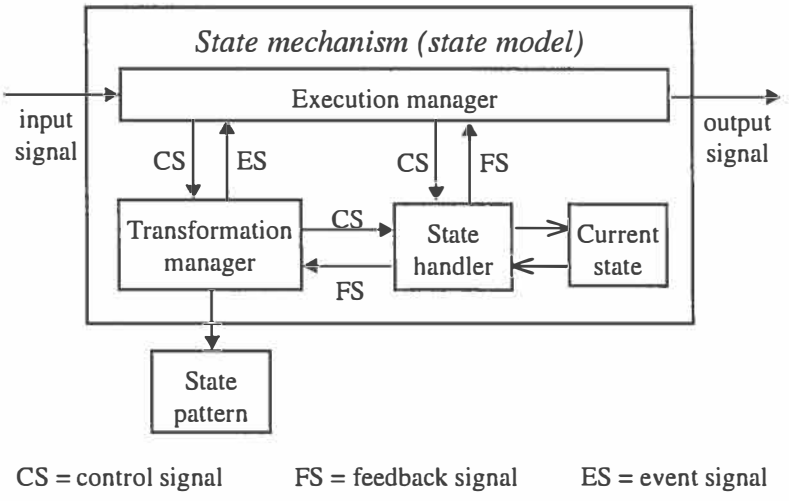

FIGURE 7 State mechanism component of the Process Engine

The state of an element node restricts what can be done with the process element at each time. Mainly, it concerns whether the process element can be

executed but it also describes the reason for its availability or unavailability: idle, available, active, finished, or exception. The current implementation of CPME uses a fixed transition model with refinable state names. The state mechanism is illustrated in figure 7.

The execution manager of the state mechanism manages a transformation manager and a state handler. The transformation manager takes care of the enactment of the state model. It evaluates each transformation based on the current state and the event that raised the transformation. The state handler takes care of all access and updates to the variable that stores the enactment state. All requests for current state information managed by the state handler.

Value mechanism. The value mechanism controls the handling of a property value according to a value pattern. It can store simple data values, references to data objects, and collections of data values or objects. Most values are entered into the system manually and the role of the value mechanism is to ensure the correctness of these values. Some values are automatically calculated or evaluated. The value mechanism supervises any automatic calculation of data values and evaluation of value-based constraints. The value mechanism is illustrated in figure 8.

The execution manager of the value mechanism controls an evaluation manager and a value handler. The evaluation manager creates a runtime instance of an action with the property value as its product and orders the function mechanism to execute the action. As the operation is finished and the result of the operation is stored, it exterminates the instance. The value handler is responsible for all access and updates to the variable that stores the property value. All requests for the value are managed by the value handler.

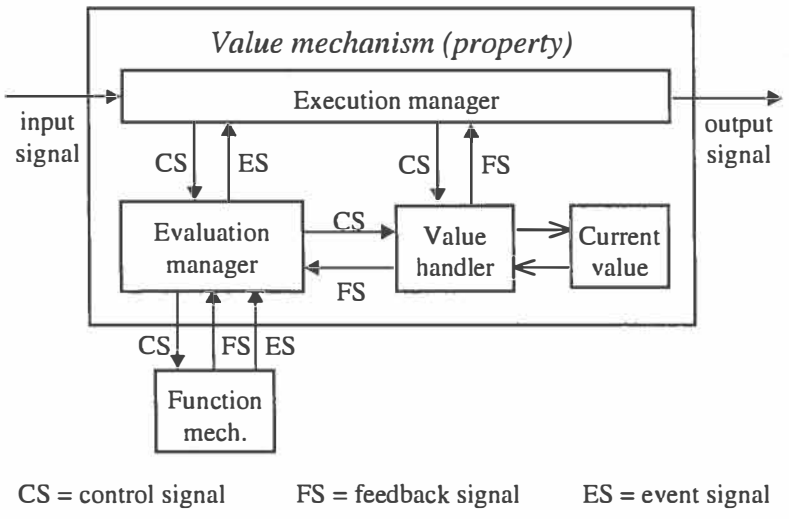

FIGURE 8 Value mechanism component of the Process Engine

2.4.4 Suggestions for Further Improvement of the Enactment Mechanisms

There are several suggestions for improving the enactment mechanisms of the Process Engine.

Firstly, we should study how the distinction between advancement and co-ordination features affects the structure of the feature handler. In the current version, a feature handler specifies – for each feature – a set of keywords and the corresponding signal processing methods. When executed, the handler checks which keyword is currently given in the specification and then executes the correct method. The current co-ordination features can be executed in the same way. However, co-ordination features might need a more sophisticated handler, since the feature handler should manage the co-ordination of several execution threads. Hence it might need a suitable checking mechanism.

Secondly, new mechanisms are needed for the detection of external events and for exchange of data across system borders. The detection mechanism should be able to detect when something specific happens in the external system, and then send an event signal to the internal system. The data exchange mechanism should be able to interpret a transformation schema for a mapping between data formats, and to execute the transformation. Version management will be an important theme.

Thirdly, we need to improve the state mechanism to make it suitable for evaluating customisable state models. This requires at least a more elaborate state manager. The execution manager is also likely to need enhancement to cope with the changed event handling system. The state handler will instead need simplification since it will no more need to handle renamed states.

Fourthly, new function mechanisms should be implemented. An envelope mechanism should control externally executed 'black-box' operations. The true controllability of such operations is very low, and therefore some manual input may be needed to track their progress. Thus the envelope mechanism requires suitable handlers for prompting and processing such manual input. Controlled function mechanisms are required for reflexive and non-reflexive 'white-box' execution. The reflexive function mechanism should be able to directly reference the process model in which it locates. It should be able to "find" its product relative to its own location in the model. This is necessary, for example, for 'token manipulation' during the execution of Petri-net based process models. The currently existing function mechanism should be enhanced to make it more suitable for non-reflexive controlled functions. This requires an improved execution controller for negotiating with tools. Further, a logic function mechanism is needed for compound logical operations on complex property values. It should enable, at least, the composition of the most basic logical operations such as negation, conjunction, and disjunction. These are needed for evaluating complex constraints.

Fifthly, the value mechanism needs to be adapted to the handling of complex values. Currently the value handler is able to manage only one value, but it should be enhanced so that it can handle any number of values. The value handler should also be modified so that it can calculate the value of the complex property from the values of sub-properties. Furthermore, the

evaluation manager should be able to use the logic function mechanism. The evaluation manager can already handle both the produced value and information of the success of the operation. Thus it will not be difficult to extend the manager to deal with feedback from the logic function mechanism. The feedback constitutes of a similar signal but only logically modified.

3 A Domain Framework for Customisable Method Support Environments

A customisable method support environment is a design aid environment. The issues addressed by a framework therefore concern various design aid functions that the environment specifies and implements. Henderson and Cooprider (1994) classify different aspects and functions of a design aid environment. They distinguish among production, co-ordination, and organisational technology. We enhance this classification in the way shown in figure 9. We reclassify production and co-ordination functionality according to whether it addresses single or multiple tasks, and single or multiple users. The resulted classification can be illustrated as follows.

	Single task	Multi-task
Single user	**"Taskware"** - production - versioning - repository	**"Processware"** - co-ordination - configuration - resources
multi-user	**"Groupware"** - user interaction - transactions - access control	**"Agentware"** - co-operation - assignment - user control
	"Helpware" - guidance - learning - trace making	- on-line material - supportive qualities

FIGURE 9 Different types of functionality in a design aid environment

"Taskware". Taskware is support technology for single tasks with a single user, and it consists of production, versioning, and repository functions. Production technology consists of support for representation, analysis, and transformation. Representation functions focus on abstraction and conceptualisation of phenomena, analysis functions reflect problem solving and decision making aspects, and transformation functions call for rules and mechanisms to

transform models. Version management introduces functions for managing design changes. Repository functions enable appropriate storage of the designs.

"Groupware". Groupware is support technology for single tasks with multiple users. It enhances taskware with functions for user interaction, transactions, and access control. User interaction concerns synchronous, concurrent use of tools, and information exchange through design information, such as attaching a note to a diagram. Transactions and access control are closely related to multi-user repository technology and its mechanisms.

"Processware". Processware is support technology for multiple single-user tasks. Although it (usually) supports multiple users, the tasks it co-ordinates engage only single users. Processware functions support task co-ordination, design configuration, and resource management. Task co-ordination allows the proper ordering and timely execution of different tasks. Configuration management aids to maintain different versions of complex system designs. Resource management enables managers to utilise project resources consistently with project goals.

"Agentware". Agentware is support technology for multiple multi-user tasks. It enhances groupware or processware by providing the users with functions for co-operation, management of assignments, and user control. Co-operation uses technology, such as electronic brainstorming and voting, to facilitate group interactions. Assignment management allows managers to allocate tasks to, and to control the work load of individual users. User control includes functionality to manage the access rights of users and user groups that participate in the system development.

"Helpware". "Helpware" aids system users in using the system and the method. It implements organisational technology. First, helpware functions help users understand and to use the design aid effectively. The functions consist of guidance (guidance support, on-line helps), learning aid, and trace making (recording project data). The system may also help enhance users' awareness of, e.g., product states, goals, peer actions, and user dependencies. Second, helpware functions establish the infrastructure on-line by providing electronic material on organisational guidelines, standard operating procedures, and quality standards. Besides, helpware introduces technology with different supportive qualities such as user friendliness and easiness.

We apply this classification on two levels in the same way as Marttiin et al. (1996). On one hand, a customisable method support environment is a design aid environment, and hence the design aid functionality is the target of customisation in the environment. On the other hand, the customisation system itself forms a design aid system (for method design) and hence the customisation system can be assessed similarly to a design aid environment.

In the following sections we introduce the domain framework and classify several criteria for the assessment of customisable method support environments.

3.1 Background to the Domain Framework

A domain framework for software process is presented by Dowson and Fernström (1994). They distinguish among three process domains. First, the process definition domain produces characterisations of software processes or process fragments using some notation. Second, the process enactment domain is concerned of enacting a process model either by humans or some automated mechanism. It uses the characterisations created in the process definition domain as input and evokes process performance as output. Third, the process performance domain encompasses the actual project activities or actions conducted by humans, and different types of supporting tools during a project. The framework does not supply criteria for system evaluation.

Koskinen and Marttiin (1998) extend this framework for customisable process modelling and support systems by introducing a fourth domain: the process metamodelling domain. In this domain, process modelling languages are created and adapted for the process modelling (cf., process definition) domain. Still, no criteria are available for system evaluation.

Elsewhere, Marttiin et al. (1996) present an evaluation framework for method engineering environments. The framework consists of two domains: customisable CASE, and CAME. The CASE domain addresses and serves multiple design aid functions (Henderson and Cooprider, 1994), whereas the CAME domain allows these functions be customised. The two-level architecture applies the same general structure in both domains, and they are thus compared and evaluated against equal criteria. Therefore, the criteria can be applied more systematically than in frameworks that have a unique view of each inspected domain. However, the weakness of the framework is that is does not consider coarse-grained process modelling and performance support.

A more comprehensive framework for customisable design environments is introduced by integrating a method engineering system and a customisable process support system (Koskinen and Marttiin, 2000). However, the framework still lacks consideration of customisable agent systems as part of the method support. Discussion of integrated meta-data models, activity models, and agent models can be found in Marttiin et al. (1995). Agent models are understood as defining the access to and use of IS models during the development tasks. They encapsulate the operations (such as querying), control (access rights, access control), and co-ordination of tasks available for the users in different roles.

We choose the framework presented by Dowson and Fernström (1994) as a baseline for our domain framework, and apply it to the framework for customisable design environments (Koskinen and Marttiin, 2000). This is further extended with aspects related to agents. The framework distinguishes among three domains in a similar way as Dowson and Fernström: a method definition domain, a method enactment domain, and a performance domain. Each domain further consists of three systems that address the three aspects of method specification and support (Marttiin et al., 1995): techniques, processes, and agents.

The domains and systems together with their interdependencies are shown in figure 10. Technique specifications, process models, and agent specifications are created in the method definition domain. They are further instrumented in the method enactment domain for to support method use in performance domain.

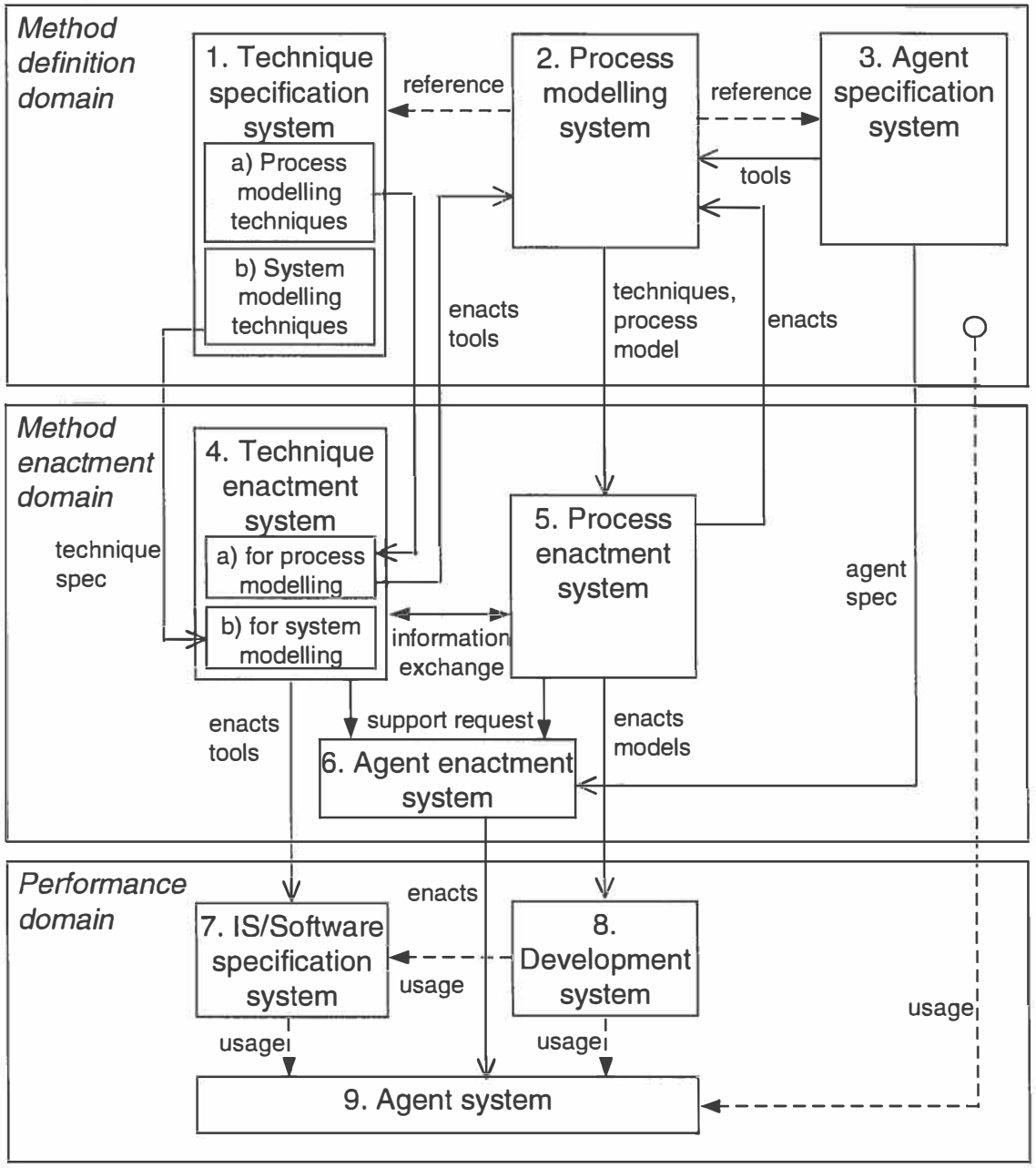

FIGURE 10 The domains of a customisable method support environment

In regard to figure 9, the IS/software specification system covers taskware, and the development systems covers processware. The agent system covers groupware and agentware. Helpware can be implemented by any of the systems. We discuss each domain and system individually in the following

sections. Several criteria are identified and collected to assess the scope, depth, and flexibility of method customisation and support functionality.

The criteria concern the coverage of the method customisation and support functions from different perspectives. Firstly, it is naturally a greater shortcoming to not have some function at all than to not have that function customisable. Therefore, many of the proposed criteria look simply for the functions that are covered in a method support system. Secondly, some criteria concern the customisation capabilities in specification and change. We are interested in what can be specified, and how much variation is allowed therein, as well as in what can be changed afterwards, and how fine the changes can be. Moreover, integration between systems is considered in that how much effort it involves to make changes in one system effective in another system. Thirdly, we consider the flexibility of method support for humans. These criteria study what kind of support is available for human performance, and how extensively it can be adapted to special needs and preferences. The forms and degree of human involvement enabled in method enactment are also addressed. Thirdly, we consider the making and adoption of changes from a human perspective. Issues in making changes include how easily and quickly changes can be made, and what kind of support there is for making the changes. Issues in adoption concern how changes interfere human performance, how people are made aware of changes, and what kind of support there is for their adoption.

3.2 Method Definition Domain

The method definition domain constitutes the method design and customisation facilities in a method support environment. It includes three systems (see figure 10): a technique specification system (1), a process modelling system (2), and an agent specification system (3). The technique specification system has two subsystems: for process modelling techniques (1a) and system modelling techniques (1b). General criteria for assessing these systems are the following:

TARGET OF REPRESENTATION
1. Architectural components
2. Representation components: structural, operational
3. Operational scopes: functions addressed in representation
4. Component characteristics
   a) Generality of underlying customisation architecture
   b) Granularity of storage and locking
   c) Forms of abstraction and reusability
   d) Component alternatives

TOOL SUPPORT
1. Tool characteristics for representation
   a) Representation style
   b) Complementary views
   c) Guidance

2. Other support functions
   a) Taskware: production, versioning, change, repository
   b) Processware: co-ordination, configuration, resources
   c) Groupware: user interaction, transactions, access control
   d) Agentware: co-operation, assignment, user control
   e) Helpware: guidance, learning, trace making, awareness

The criteria are divided in three main groups: the target of representation, the representation support, and other support functions. First, the criteria for the target of representation concern the scope of components that can be represented, and the terms and notations in which a method can be represented. Also the support functions that can be addressed are considered, as well as several other characteristics that improve the definition mechanisms and the adaptability of methods.

Second, the criteria for tool support are concerned with the characteristics of method customisation tools. The representation style specifies in which forms method definitions can be represented and modified (e.g., textual, graphical), and the complementary views allow alternative modes or perspectives to represent the methods. Since method definition is often a complex task, also guidance for method definition is considered.

Third, other support functions may address any of the design aid functions shown in figure 9.

The general criteria are applied in the same manner in each definition system. The differences can be found regarding the target of representation. We discuss the application of the general criteria in each definition system.

3.2.1 Technique Specification System

A technique specification system defines different representation, analysis, and transformation techniques either for system modelling or process modelling. In the latter case, the system corresponds to the "process metamodelling domain" in the framework by Koskinen and Marttiin (1998). The specific criteria for technique specification systems are the following:

ARCHITECTURAL COMPONENTS
1. Model of representation styles
2. (Process) meta-metamodel: metatypes and dependencies

REPRESENTATION COMPONENTS
1. Representation styles (views, visual fragments, visual attributes)
2. Conceptual framework: conceptual construct and dependency types
3. Notation: notational construct and dependency types
4. Execution semantics: semantic construct and dependency types
5. Operational semantics: operational construct and dependency types

OPERATIONAL SCOPES
1. Taskware: representation, analysis, transformation, versioning, change
2. Helpware: guidance, learning, trace making, awareness

COMPONENT CHARACTERISTICS
1. Generality of the (process) meta-metamodel and the model of styles
2. Granularity of storage and locking for representation components
3. Forms of abstraction and reusability in specification
4. Alternative representation styles, notations, and semantics

TOOL SUPPORT

The specific criteria are divided into four groups: architectural components, representation components, operational scopes, and component characteristics. First, the architectural components addressed are the model of representation styles that determines the range of possible notations, and the (process) meta-metamodel that determines what can be specified. Second, the target of representation is refined to cover different parts of a technique: the language (conceptual framework, notation, execution semantics) and the operational semantics of the technique. Third, the operational scope is refined to functions that a technique may address in taskware and helpware. Fourth, component characteristics are refined to concern specific representation components. Tool support, however, is assessed according to the same general criteria.

3.2.2 Process Modelling System

A process modelling system defines and adapts process models. In the framework by Dowson et al. (1994) this corresponds to the process definition domain. The system uses the process modelling techniques and tools supplied by the technique specification system and uses them to produce process models. The specific criteria for process modelling systems are the following:

ARCHITECTURAL COMPONENTS
1. Process metamodel: process types and dependencies
2. Representation framework: representation types and dependencies
3. Use of a customisable process modelling technique

REPRESENTATION COMPONENTS
1. Conceptual process model: process concepts and dependencies
2. Process representation: perspectives, representations and dependencies

OPERATIONAL SCOPES
1. Processware: co-ordination, configuration, resources
2. Helpware: guidance, learning, on-line material, trace making, awareness

COMPONENT CHARACTERISTICS
1. Generality of the process metamodel and the representation framework
2. Granularity of storage and locking for process models and perspectives
3. Forms of abstraction and reusability in process models
4. Alternative process structures, perspectives, and representations

TOOL SUPPORT

The specific criteria are divided into four groups: architectural components, representation components, operational scopes, and component characteristics. First, the architectural components addressed are the process metamodel that determines what can be specified, and the representation framework that determines the representation types and dependencies. In case the system uses a customisable process modelling technique, the architectural components are not as restricting. Second, the target of representation is refined to cover the conceptual process model and process representations and perspectives. Third, the operational scope is refined to functions that a process model may address in processware and helpware. Fourth, component characteristics are refined to concern the specific representation components. Further, tool support is assessed according to the general criteria.

3.2.3 Agent Specification System

The agent specification system defines agent interactions, transactions, and control. The system addresses co-ordination functions both in the process modelling system and the development system. The specific criteria for agent specification systems are the following:

ARCHITECTURAL COMPONENTS
1. Generic agent profile architecture: generic profile types and dependencies
2. Agent metamodel: agent types and dependencies
3. Representation framework: representation types and dependencies

REPRESENTATION COMPONENTS
1. Agent profile types
2. Conceptual agent model: agent concepts and dependencies
3. Agent representation: perspectives, representations
4. Operation architecture: operational constructs and dependencies

OPERATIONAL SCOPES
1. Groupware: user interaction, transactions, access control
2. Agentware: co-operation, assignment, user control
3. Helpware: guidance, learning, on-line material, trace making, awareness

COMPONENT CHARACTERISTICS
1. Generality of the profile architecture, the agent metamodel and the representation framework
2. Granularity of storage and locking for profiles, agent concepts, and agent perspectives
3. Forms of abstraction and reusability in agent specification
4. Alternative agent structures and perspectives

TOOL SUPPORT

The specific criteria are divided into four groups: architectural components, representation components, operational scopes, and component characteristics. First, the generic agent profile architecture determines the degree to which there can be variation in agent profile structures. The agent metamodel determines what can be specified of agents, and the representation framework determines the types of agent representation. Second, the target of representation is refined to cover the agent profile types, conceptual agent models, and agent perspectives, and representation. Third, the operational scope is refined to the functions that an agent specification may address in groupware, agentware, and helpware. Fourth, component characteristics are refined to concern the specific representation components. Further, tool support is again assessed according to the general criteria.

3.3 Method Enactment Domain

The method enactment domain constitutes the method instrumentation facilities in a method support environment. The systems included are (see figure 10): the technique enactment system (4), the process enactment system (5), and the agent enactment system (6). General criteria for method enactment systems are the following:

TRANSFORMATION FROM DEFINITION TO ENACTMENT
1. Integration to the definition system
   a) Means of system integration and the mappings needed
   b) Means of mapping: transformation / interpretation / execution
   c) Time of mapping: pre-execution / runtime
   d) Granularity of mappings
2. Runtime method changes
   a) Support for testing: runtime simulation, prototyping
   b) Support for automation: reflection
   c) Support for management: change propagation, state rebuilding

ENACTMENT ARCHITECTURE
1. Multi-user architecture
   a) Multi-user components
   b) Component interaction

2. Enactment mechanism
   a) Components of the mechanism
   b) Component characteristics: generality, granularity
3. Enactment tasks

INTEGRATION TO THE PERFORMANCE SYSTEM
1. Generating method support
2. Interaction with the performance system

The criteria are divided into three main groups: transformation from definition to enactment, enactment architecture, and integration to the performance system. First, the criteria for transformation from definition to enactment concern the integration of the enactment system to the definition system, and runtime method changes. Integration to the definition system addresses the means of integration and the mappings needed. The mappings are based either on transformation, interpretation, or execution. The mapping may be created before execution, or at runtime. There is also variation in the granularity of mappings: fine-grained mappings allow more flexibility than coarse-grained mappings in method change. Fine-grained runtime mappings are required for incremental modification of methods. The support for runtime method changes is also considered: the forms of support for testing, propagating, and managing changes.

Second, the enactment architecture is addressed from three aspects. The distribution of enactment mechanisms in a multi-user architecture and interaction between the components are studied. The components of the components are considered, and their generality and granularity. The generality of components contributes directly to the range of method variation, while the granularity contributes to the flexibility of method change. The variety of tasks that the enactment mechanism may cover is dependent on the specific enactment system and will be discussed in connection to the specific systems.

Third, the way in which the enactment system is integrated to the performance system is considered from the viewpoint of how method support is generated, and how the enactment mechanism interacts with the performance system.

3.3.1 Technique Enactment System

A technique enactment system instruments techniques for their use in the IS/software specification system (or the process modelling system). The specific criteria for technique enactment systems are the following:

TRANSFORMATION FROM DEFINITION TO ENACTMENT
1. Integration to the definition system
2. Runtime technique changes

ENACTMENT ARCHITECTURE
1. Multi-user architecture
2. Enactment mechanism
   – Mechanisms for operation on the specification components

ENACTMENT TASKS
1. Construction of specifications
   a) construction of conceptual specification components
   b) construction of representations
2. Coverage of other taskware functions
   a) Executing analyses of specifications
   b) Executing transformations on specifications
   c) Creating and managing specification versions
   d) Controlling specification changes
3. Coverage of help functions

INTEGRATION TO THE PERFORMANCE SYSTEM
1. Building tool support: assembly, mediation, generic tool architecture
2. Co-ordination with fixed tool operations

The criteria are divided into four groups: transformation from definition to enactment, enactment architecture, enactment tasks, and integration to the performance system. The first two groups of aspects are assessed according to the general criteria, except that the components of the enactment mechanisms are refined to the mechanisms for operating on the specification components. The enactment tasks consist of construction of specification, and coverage of other taskware and help functions. First, the construction of specification is concerned with the construction of conceptual specification components, and representations. Second, the taskware functions may include executing analyses and transformations, creating and managing specification versions, and controlling specification changes. Different helpware functions may also be covered.

Furthermore, system integration is considered from the viewpoint of building tool support, and co-ordinating with fixed tool operations. First, tool building may be based on assembly, mediation, or generic tool architecture. These allow different forms and levels of customisability. Second, there are always some "fixed" operations in customisable tools. The integration should be based on a mapping between the fixed "interface layer" and the metatypes of the (process) meta-metamodel. The "thinner" the interface layer is, the more extensive adaptation it allows.

3.3.2 Process Enactment System

A process enactment system manages the automated enactment of process models supplied by the process modelling system. In the framework by Dowson et al. (1994) this corresponds to the process enactment domain. The specific criteria for process enactment systems are the following:

TRANSFORMATION FROM DEFINITION TO ENACTMENT
1. Integration to the definition system
2. Runtime process model and process metamodel changes

MULTI-USER ARCHITECTURE

PROCESS ENGINE
1. Components of the process engine
   a) Mechanisms for interpreting a process metamodel
   b) Mechanisms for interpreting process model components
   c) Mechanisms for executing process components
2. Component characteristics

ENACTMENT TASKS
1. Progress along lateral and hierarchical channels
2. Tool invocation and execution control
3. Controlling changes in process data and their effects on enactment
4. Controlling the evolution of enactment states
5. Detecting and informing about external events
6. Controlling automated data exchange, import and export

GENERATING METHOD SUPPORT
1. Types of support: passive, guiding, restricting
2. Variation in the type of support

INTERACTION WITH THE PERFORMANCE SYSTEM
1. Information exchange with tools
2. Controlling interaction between tools

The criteria are divided into six groups: transformation from definition to enactment, multi-user architecture, process engine, enactment tasks, generating method support, and interaction with the performance system. The first two groups of aspects are assessed according to the general criteria. However, runtime changes concern both process models and process metamodels. Second, the components of the process engine may include mechanisms for interpreting a process metamodel, for interpreting process model components, and for executing process components. The component characteristics are assessed according to the general criteria.

Third, the tasks of a process engine consist of interpreting a process model and managing its execution. These tasks may include 1) to manage progress along lateral and hierarchical execution threads in the process model, 2) to manage tool invocation, 3) to control changes in variable values (data properties and constraints) and the effects of these changes on enactment, 4) to control the evolution of enactment states, 5) to detect and inform about external events, and 6) to control data exchange, import and export (Koskinen, 2000a).

Fourth, we consider the types of support. Passive support must be requested by the user. In guiding mode, the system attempts to detect

situations in which it should offer help automatically, whereas in restricting mode, the system controls performance either by disabling or enforcing operations. The variation in support is also considered.

Fifth, the form of information exchange between the process engine and the tools are considered. There may be a separate tool broker, or the tools themselves may be capable of negotiating with the enactment mechanism. There is also variation in whether the enactment system has control on the interaction between tools.

3.3.3 Agent Enactment System

An agent enactment system manages the co-ordination of agents specified by the agent specification system. The specific criteria for agent enactment systems are the following:

TRANSFORMATION FROM DEFINITION TO ENACTMENT
1. Integration to the definition system
2. Runtime agent and profile changes

MULTI-USER ARCHITECTURE

AGENT ENGINE
1. Components of the mechanism
   – Mechanisms for executing agent components
2. Component characteristics

ENACTMENT TASKS
1. Managing fine-grained user interactions between tools
2. Managing user transactions and task transactions
3. Requesting different types of access control
4. Managing coarse-grained information exchange
5. Evaluating agent profiles
6. Managing assignments and enforcing user rights

GENERATING AGENT SUPPORT
1. Building tool support
2. Forms of tool support

INTERACTION WITH THE PERFORMANCE SYSTEM
1. Monitoring agent behaviour
2. Restricting agent behaviour
3. Information exchange with agents

The criteria are divided into six groups: transformation from definition to enactment, multi-user architecture, agent engine, enactment tasks, generating agent support, and interaction with the performance system. The first two groups of aspects are assessed according to the general criteria. Runtime changes, however, concern agent and profile changes. Second, the components

of the agent engine include mechanisms for executing agent components. The component characteristics are assessed according to the general criteria.

Third, the tasks of an agent engine may include 1) to manage fine grained user interactions between tools, 2) to manage user transactions and task transactions, 3) to request different types of access control, 4) to manage coarse-grained information exchange, 5) to evaluate agent profiles, and 6) to manage assignments and enforcing user rights.

Fourth, generating agent support is concerned with building tool support, and the forms of the tool support. Tool support incorporates various interfaces for interaction with other agents, and agent-specific views of, e.g., work, products, and resources. Interaction with the performance system includes monitoring and restricting agent behaviour, and information exchange with agents.

3.4 Performance Domain

The performance domain constitutes the method support facilities in a method support environment. It includes three systems (see figure 10): the IS/software specification system (7), the development system (8), and the agent system. The general criteria for performance systems are the following:

FLEXIBILITY OF SUPPORT
1. Variation of support
2. Variation of perspectives

SUPPORT FOR HUMAN ENACTMENT
1. Degree of human enactment
2. Providing enactment information

INTRODUCING METHOD CHANGES
1. Level of interference
   a) Restricted change propagation
   b) Granularity of exception locks
2. Support for awareness
   a) Notification
   b) "Notice board"
3. Support for adopting changes
   a) Guidance
   b) Learning aid
   c) Tracing

The criteria are divided into three main groups: flexibility of support, support for human enactment, and introducing method changes. First, flexibility of method support can be introduced by variation of support and perspectives. Second, support for human enactment as the degree of human involvement in enactment, and the level of provided enactment information.

There are several concerns related to the introduction of method changes. First, the level of interference on performance caused by method changes can be diminished by restricted change propagation that allows users to choose the time when changes are introduced. The granularity of exception locks is a significant contributor to the level of interference. The smaller the size of the lock is, the less the change is likely to interrupt the users. Second, there are different ways to support awareness of changes. These include automatic notifications and the use of an electronic "notice board". Third, support for adopting changes may include guidance, learning aid, and support for tracing changes.

There are no significant differences between the performance systems regarding these criteria.

4 Assessment of the MetaEdit+/CPME Implementation

We assess how the current MetaEdit+/CPME implementation addresses different aspects of the domain framework using the developed criteria. To allow a more consistent presentation, we organise the discussion of each system by discussing it in connection to the relevant set of systems (definition, enactment, performance). We discuss systems for system modelling techniques (Section 4.1), systems for process modelling techniques (Section 4.2), systems for processes (Section 4.3), and systems for agents (Section 4.4).

4.1 Systems for System Modelling Techniques

4.1.1 Technique Specification System

Target of representation. The meta-metamodel of MetaEdit+ is GOPRR (Kelly, 1998). The five conceptual metatypes (Graph, Object, Property, Role, Relationship) are used to specify different types of conceptual language constructs, and there are several generic dependencies possible between them (inclusion, decomposition, explosion, attribution, object link, binding). The GOPRR metatypes are instantiated to form conceptual metamodels. The conceptual construct types can be fully customised, but dependency types only partially.

MetaEdit+ does not support the modification of representation styles, but it maintains three built-in representation styles: diagrams (that consist of symbols, lines, and labels), matrices, and tables. Notations are currently specified only to a degree that is necessary to give different concepts a specific representation. The specification of conceptual frameworks and notations is equally acknowledged in the GOPRR model, but conceptual frameworks are emphasised over notations in its current implementation.

Semantics for simulation or semantics cannot be specified. However, joint research on the former is currently initiated. Also, operational specification is

not supported. The system implements a generic GOPRR specific process pattern that applies to any technique in the system. The pattern is extended by the style-specific process patterns implemented by the generic modelling tools. The style-specific process pattern applies to any technique using the specific style. A joint research effort on integrating metamodelling and guidance modelling has yielded some theoretical results (Lyytinen et al. 1998). The guidance system in concern is presented by Pohl et al. (2000).

The GOPRR model does not enforce a specific system ontology. The modelling technique components (each instance of the metatypes) are individually stored in a repository. Different notations within a representation style are not possible, but simultaneous use of the three representation styles for a specific technique is not restricted. From the perspective of reuse, the lack of alternative notations of the same style is awkward since it restricts the reusability of concept types. MetaEdit+ supports specialisation and component reuse for technique specification (Rossi, 1998; Zhang, 2000).

Tool support. The CAME tool set includes form based tools for creating and managing metamodel components and composing them into technique specifications. Other representation styles or complementary views to techniques are not supported. Apart from informing when the user attempts to violate the GOPRR rules, no specific guidance for metamodelling is provided.

MetaEdit+ provides basic tools for the representation of modelling techniques. No support for analysis and transformation of techniques is currently available. Versioning of technique components is not supported. All changes to techniques are not automatically propagated on the models, and hence there is some protection against unmanaged changes. In case a modification would cause irreversible changes (such as deletion of properties), the metamodeller is warned.

Co-ordination of coarse-grained technique specification processes can be supported by CPME. The agent system controls metamodelling rights. Currently it allows only one metamodeller modify technique specifications at a time.

4.1.2 Technique Enactment System

Transformation from definition to enactment. The technique enactment system is integrated to the specification system by incrementally compiling the technique specification components into executable MetaEngine components. The integration allows runtime mapping by automatically updating the MetaEngine when any of the components is changed. The granularity of mappings is fine-grained: property types are compiled independent of the object types to which they are attributed, and vice versa. Consequently, MetaEdit+ also supports prototyping of techniques. As discussed above, change propagation is only partial.

Enactment architecture. MetaEdit+ is implemented on a client-server architecture. Clients do not communicate with each other directly, but through the shared design information. Technique enactment in MetaEdit+ is managed

by the MetaEngine (Kelly et al., 1996). The MetaEngine embodies the implementation of the underlying conceptual data model, GOPRR, and its operation signature. This operation signature covers the generic process pattern used by all techniques. Each client in the multi-user architecture has a copy of the MetaEngine. MetaEdit+ distinguishes between operations on model representations managed in tools and operations. The MetaEngine manages all operations on the underlying conceptual data through its service protocol. The generality of the MetaEngine components is high, since they implement the generic GOPRR model. For the same reason, they have are fine-grained.

The construction of conceptual specification components is managed by the MetaEngine, whereas the construction of representational data is managed by the generic modelling tools. Manipulation of the conceptual data is always requested from the MetaEngine. User-customisable analyses of system models, and transformations into textual documents can be made by means of reporting. MetaEdit does not support versioning of system models. Support for specification changes is rudimentary. Additional help is not included.

Integration to the performance system. MetaEdit+ contains several generic tools for system modelling that are configured simply by supplying them with a metamodel. As noted above, the modelling tools implement a generic process pattern that is used by any modelling technique. Different operational versions of the tools are therefore not possible. Since operational variation is not possible, the level of enforcement is restricted to a flexible one. The system does not allow the creation of illegal constructs and combinations, but compliance to some rules, especially after changes, is difficult to ensure without adequate, automated checking functions.

4.1.3 IS/Software Specification System

Flexibility of support. The variation of support provided by MetaEdit+ concerns the conceptual and representational modifiability of the representation system and the related reporting and query functionality. Operational modifications cannot be done. Variation of perspectives is provided by introducing three representation styles: graphical, matrix, and tabular. Representation independence is also supported, and hence one conceptual model may have several complementary and overlapping representations.

Support for human enactment. The modelling tools are based on human enactment. No enactment information is automatically collected or provided.

Introducing method changes. MetaEdit+ supports restricted change propagation. No exception locks are used. Tools that are being used while a change in the current technique is introduced, do not automatically respond to the change. The changes are updated either by refreshing the tool window (changes in the notation), or closing and reopening it (conceptual changes reflected in the toolbars and menus). Automatic notification is not supported, nor there is a "notice board" for changes. MetaEdit+ does not provide help for adopting changes in techniques.

4.2 Systems for Process Modelling Techniques

4.2.1 Technique Specification System

Target of representation. The process meta-metamodel of CPME is GOPRR-p. The GOPRR-p model is discussed in detail in Section 2.3. CPME does not support the modification of representation styles. Similarly to GOPRR, conceptual frameworks are currently more emphasised than notations. Diagrams are the core representation style used for process modelling, but process models can be viewed and edited also in matrix and table forms.

The major difference between metamodelling in MetaEdit+ and process metamodelling in CPME is that the latter extends the metamodelling system with mechanisms for specifying enactment semantics. The semantics can be applied also in simulation. A detailed study is conducted that distinguishes among conceptual, notational and semantic process metamodels (Koskinen, 2000a). The presented design allows each aspect to be specified independently.

Operational specification is not supported. However, some theoretical considerations on the subject have been presented (Koskinen, 2000a). The study introduces operational modelling as a means for operational specification. It is stated that operational modelling should address not only modelling operations, guidance and tracing, but also reuse, configuration and versioning, changes, and transactions. In brief, it should cover everything that intimately relates to modelling with the technique and that should be performed when the technique is used.

The GOPRR-p model does not enforce a specific process ontology. The technique components (each instance of the metatypes) are individually stored in a repository. Specialisation and reuse are supported. Similarly to GOPRR, different notations within a representation style are not possible, but the use of the three representation styles for a specific technique is not restricted.

Tool support. CPME includes a set of form based process metamodelling tools that are used to create and change process metamodels (Koskinen and Marttiin, 1997). The current implementation of process metamodelling tools and the process metamodel architecture is largely based on the implementation of metamodelling tools and a metamodel architecture in MetaEdit+. Complementary views to techniques are not supported. No guidance for process metamodelling is available apart from informing of attempted violations against the GOPRR-p rules.

MetaEdit+ provides basic tools for the representation of process modelling techniques. No support for analysis and transformation of techniques is available. Versioning of method components is not supported. Co-ordination of metaprocesses can be supported by CPME itself. The agent system allows several process modellers work on process models simultaneously. It also controls the process modelling rights.

CPME lacks mechanisms to track changes made to system modelling techniques in the technique specification system. Thus it cannot provide

automated support for detecting metamodel changes that might affect the development process and thus the process model.

4.2.2 Technique Enactment System

Transformation from definition to enactment. Integration of the specification and enactment systems is realised in the same way as for system modelling techniques. The core of the enactment system is the Process Engine that implements the GOPRR-p model (Koskinen and Marttiin, 1997). Technique specification components are incrementally compiled, and the Process Engine is automatically updated when any of its components is changed. The granularity of mappings is similarly fine-grained, and change propagation partial.

Enactment architecture. In process modelling, the Process Engine plays the same role as the MetaEngine in system modelling. Hence the discussion of the MetaEngine and the multi-user architecture applies also to the Process Engine. However, the conceptual data model that the Process Engine implements and its operation signature extends beyond the manipulation of model components. The Process Engine shares the task of constructing process model with the Process Editor. The Process Engine manages the construction of conceptual specification components, and provides the Process Editor with information of the notation. The Process Editor manages the construction of process representations.

Integration to the performance system. The process modelling tools in CPME are almost direct extensions to the system modelling tools in MetaEdit+. Similarly, the tools are configured by supplying them with a process metamodel. Different operational versions of the tools are not possible. Level of enforcement in a supported process is restricted to a flexible one.

4.3 Systems for Processes

4.3.1 Process Modelling System

Target of representation. CPME supports three representation frameworks simultaneously: graphical, matrix, and tabular. However, process models in CPME are primarily graphical and concentrate on process elements on the coarse-level of task co-ordination. CPME allows several conceptual perspectives and several representations for a conceptual process model. The operational scopes addressable in process models depend on the underlying performance functionality: process support is based on the co-ordination of this functionality. Since MetaEdit+ does not support configuration of system specifications, nor handling of resources, they cannot be addressed in process models, either.

As discussed above, process metamodels in CPME are customisable. Therefore the generality of the process model components is high. The granularity of storage and locking of process model and representation components is also fine-grained. All GOPRR-p based components are stored

automated support for detecting metamodel changes that might affect the development process and thus the process model.

4.2.2 Technique Enactment System

Transformation from definition to enactment. Integration of the specification and enactment systems is realised in the same way as for system modelling techniques. The core of the enactment system is the Process Engine that implements the GOPRR-p model (Koskinen and Marttiin, 1997). Technique specification components are incrementally compiled, and the Process Engine is automatically updated when any of its components is changed. The granularity of mappings is similarly fine-grained, and change propagation partial.

Enactment architecture. In process modelling, the Process Engine plays the same role as the MetaEngine in system modelling. Hence the discussion of the MetaEngine and the multi-user architecture applies also to the Process Engine. However, the conceptual data model that the Process Engine implements and its operation signature extends beyond the manipulation of model components. The Process Engine shares the task of constructing process model with the Process Editor. The Process Engine manages the construction of conceptual specification components, and provides the Process Editor with information of the notation. The Process Editor manages the construction of process representations.

Integration to the performance system. The process modelling tools in CPME are almost direct extensions to the system modelling tools in MetaEdit+. Similarly, the tools are configured by supplying them with a process metamodel. Different operational versions of the tools are not possible. Level of enforcement in a supported process is restricted to a flexible one.

4.3 Systems for Processes

4.3.1 Process Modelling System

Target of representation. CPME supports three representation frameworks simultaneously: graphical, matrix, and tabular. However, process models in CPME are primarily graphical and concentrate on process elements on the coarse-level of task co-ordination. CPME allows several conceptual perspectives and several representations for a conceptual process model. The operational scopes addressable in process models depend on the underlying performance functionality: process support is based on the co-ordination of this functionality. Since MetaEdit+ does not support configuration of system specifications, nor handling of resources, they cannot be addressed in process models, either.

As discussed above, process metamodels in CPME are customisable. Therefore the generality of the process model components is high. The granularity of storage and locking of process model and representation components is also fine-grained. All GOPRR-p based components are stored

independently. Reuse mechanisms for component cloning and sharing (component reuse) are supported. CPME allows the specification of alternative process structures, perspectives (limited regarding notations), and representations.

Tool support. The Process Editor in CPME is a generic, graphical process modelling editor that is configured by supplying it with a process metamodel. Thus it enables the use of different, adapted process modelling languages. Complementary and overlapping representations of a conceptual process model can be used. The Process Editor does not provide guidance for process modelling.

A process model can be inspected through simulation. The Process Editor aids this inspection by changing symbol colours according to the enactment state. Manual consistency checking is possible by running reports using the Report Editor. The reports are metamodel specific and they defined using the Report Editor. The reporting functionality is based on model transformation into a textual format. It can be used for ordinary reports but also document and code generation (e.g., HTML, programming languages). Thus, CPME (or MetaEdit+ alone, since it provides the same capability) could also be used as a front-end design environment for generating formal process definitions for an external process engine.

The functionality of CPME can be used also for metaprocess support. Process modelling is allowed only for users having process modelling rights in the project.

4.3.2 Process Enactment System

Transformation from definition to enactment. In CPME, no transformation between the process modelling system and the process enactment system is needed. Process models are built of generic components that are able to execute themselves. The components can be organised into complex hierarchical and network structures (or mixed).

Process model "templates" can be "instantiated" by cloning, and any process model may act as a template. However, changes to the "templates" are not propagated to the instantiations. Instead, the process metamodel can be used for such changes.

Since the Process Editor is used both in process modelling and as an interface between process enactment and process performance, the system supports rapid prototyping. Simulation is carried out otherwise similarly to enactment, but functions that would change the state of the performance system are not executed. CPME also supports reflection.

Enactment architecture. Each of the clients on the multi-user architecture has also a copy of the Process Engine. All process data are stored in the repository, including the enactment states for each process element. The components of the Process Engine are characteristically general and fine-grained. Process models are constructed of small customisable constructs that have a common generic structure with low coupling. The Process Engine

encapsulates all model components in a specific enactment mechanism, which results in a highly flexible, executable system. Execution of process models is based on a simple message exchange system, in which messages are forwarded and manipulated along the model structure (see Section 2.4.3).

The Process Engine manages progress along lateral and hierarchical execution threads, and controls tool invocations, changes to property values and their effects, and evolution of enactment states. Instead, interfaces to external systems, including event detection, and data exchange, import and export are not implemented.

Integration to the performance system. The Process Engine uses the function mechanism (see Section 2.4.3) as an interface to the performance system. The function mechanism invokes and controls the execution of tools in the performance system. However, it cannot control the interaction between modelling tools, since there is currently no proper interface in the tools. When a tool in integrated to the process enactment system, an appropriate interface must be coded into the tools. The process programming interface (PPI) provides help on selecting tool operations and verifying the correctness of product types both for definition of action types and at time of enactment. This is unsatisfactory from the viewpoint of CPME. Consequently, a PPI is currently implemented only in a few tools for the prototype purpose. Designing a tool interfacing system is suggested as a future research task (Koskinen, 1999).

Since MetaEdit+ does not provide the necessary tool functionality for active guidance and prompts, some additional tools are implemented to augment process support with guidance and tracing. This functionality includes different kinds of dialogs for viewing and prompting information for and from users, recording process data (such as time stamps), and checking conditions. The suite of these tools and operations can be extended according to new process support needs.

4.3.3 Development System

Flexibility of support. The variation of support provided by CPME concerns both process approaches (languages, thinking) and the conduct of a development process. The former is enabled by customisation of process metamodels, and the latter by the customisation of process models. The current implementation of CPME allows passive and guiding process support. The current design allows working without or around a process model and, if necessary, the state and the structure of an enacting process model can be later updated with results achieved apart from it. CPME does not transform a metaCASE environment into a process-centred environment, and it does not control activities carried out in MetaEdit+ tools. The metaCASE environment remains an autonomously functioning environment. Process support is provided when MetaEdit+ tools are used through CPME. Variation of perspectives is provided by representing different conceptual perspectives to the process.

Support for human enactment. Process support in CPME is coarse grained and intended for guiding and co-ordinating modelling tasks. The approach used in CPME can be characterised as computer aided human enactment. It mainly provides users with different levels of guidance (active and passive) and an interface for tool invocations. Process enforcement would require a mechanism to control the MetaEdit+ tool launcher and tool menu options that allow access to other tools. Currently, a process model can be enacted only through the Process Editor (and partially through the Matrix Editor and Table Editor). However, the actual interface to a process model is a set of menus, and therefore the enactment interface could also be added as an additional menu in system modelling tools. Information of enactment states is visualised the in the Process Editor as changing colours. The state of the selected process element is also shown textually in the information box of the editor. Available tasks can be requested from the Process Engine, as well as other information on enactment states.

Introducing method changes. The model components and the structure are fine-grained to a degree that most changes could be made without any exception lock. The underlying GOPRR-p model ensures that no model change can unintentionally halt the execution of a process model. Nevertheless, the process modeller may manually set an execution lock to ensure process consistency during a change. Despite this, the consistency cannot be fully ensured due to a lack of an appropriate state rebuilding mechanism. Such a mechanism has been designed but yet not implemented. The rebuilding mechanism is intended for managing "break up points", where a new component is inserted into an already executed section of the model, and is intended to affect the states of succeeding components.

There is currently no mechanism that would automatically inform the users about changes made to the language or the process model. However, such mechanism could be modelled in a metaprocess model. In this way, also guidance, learning aid, and tracing could be introduced at a coarse level to support the adoption of changes.

4.4 Systems for Agents

4.4.1 Agent Specification System

Target of representation. The agent specification system in MetaEdit+ and CPME is very simple. The basic agent system allows the specification of projects, users, and user rights. Several users are assigned to a project and one user may participate in several projects simultaneously. Each user is specified with a name and a unique login name. Administrator rights and metamodelling rights are project-specific. CPME extends the basic system with user roles, and a capability to assign tasks for user roles. User roles are specified as simple names. Process modelling rights are project-specific.

Tool support. Since the agent system is rudimentary, MetaEdit+ and CPME provide only very simple tools for agent specification.

4.4.2 Agent Enactment System

Transformation from definition to enactments. There is no explicit agent enactment system in MetaEdit+ or CPME. Instead, the agent mechanisms are encoded in the repository manager, and in the MetaEngine and the Process Engine. User information is managed by the repository manager. There are separate tools for specifying agent roles and assigning task to those roles. All agent information can be changed at runtime.

Enactment architecture. User interaction is enabled through shared design and process information stored in a common repository. Sessions and transactions are managed by the repository manager. A session consists of subsequent transactions each of which ends with committing or abandoning all changes made during that transaction. There is some inconvenience from not allowing concurrent task-specific transactions within one user transaction. MetaEdit+ supports also long transactions that cover several sessions.

The repository manager enforces access rights to the design information. Also updates to clients after commits are managed by the repository manager. Task assignment is based on assigning users to user roles, to which process elements in a process model are further assigned. Since there is no specific agent engine in MetaEdit+ or CPME, user control for task rights is enforced by the Process Engine.

Integration to the performance system. Integration is mostly achieved by having the functions fixed in the system.

4.4.3 Agent System

Flexibility of support. Variation is not provided for support nor for agent perspectives.

Support for human enactment. Access rights are fine-grained: write locks can be set for individual components in a model. Furthermore, write locks are set independently for representation components and conceptual components. Therefore, several users may modify model components simultaneously. The system does not provide enactment information on agents.

The Process Engine automatically checks whether the current user may execute a process element or not. The menus of the Process Editor are modified accordingly. All currently accessible process elements can be requested from the Process Engine. The set of process elements is automatically displayed in the pop-up menus of the Process Editor. The scope of the request is determined based on the selected model component. If no individual component is selected, all accessible process elements are provided.

Introducing method changes. The changes allowed in agents are so minor that the changes do not interfere in agent enactment.

5 Discussion

Research on metaCASE process support today is diverse and scattered, and there are no general architectures that would show direction for a unified body of relevant research. This paper attempts to contribute to this end. We have described and analysed the prototypical implementation of a generic process modelling and enactment system, CPME. It is designed as a customisable process support system for a metaCASE environment, MetaEdit+. The integrated MetaEdit+/CPME is a customisable method support environment. Furthermore, we have developed a set of assessment criteria for such customisable environments, and assessed MetaEdit+/CPME against these criteria. Through these criteria, we aim to achieve a more comprehensive view of customisable method support environments. MetaEdit+ and the CPME prototype are assessed to find ways for their further improvement.

We summarise the results of our assessment in several major themes of improvement request for MetaEdit+ and the CPME prototype. These reflect shortcomings in support functions. First, there are *general improvement* requests that concern

1) more comprehensive support for method design tasks,
2) version and change management aspects,
3) improved forms of abstraction and reuse, together with enhanced component management, and
4) improved tool support.

Second, there are requests that concern the *improvement of technique-related support*. These include

1) improved facilities for the specification of notations and representation styles,
2) support for operational specification,
3) support for the specification of simulation semantics for system modelling, and
4) support for the specification of analysis and transformation techniques.

Third, there are requests that concern the *improvement of process-related support*. These cover

1) improved multi-user support,
2) improved repository support,
3) mechanisms for event detection, data exchange, import and export,
4) improved process programming interfaces and interfacing system for tools, and
5) support for configuration and resource management.

Fourth, an *improved agent system* with specification and enactment facilities is needed.

There are also some issues in MetaEdit+/CPME design that we find particularly interesting for developers of customisable method support environments. These include

1) the MetaEngine and the integration between technique specification and IS/software specification system,
2) the specification of process modelling languages and techniques,
3) the Process Engine and the integrated process modelling and enactment system.

These demonstrate that specialisation and high-level integration between different systems is the key contributor to flexibility in customisable method support environments.

For a wider audience, the assessment of MetaEdit+ and the CPME prototype acts as an example of how to apply the criteria to a method support environment. The criteria are not designed as a checklist where one could simply check whether some feature is supported or not. We have experienced that an assessment is more fruitful when it delivers a proper description of the system under study. Thereby, the various individual and unique features of different systems can be better accounted for in the assessment. The criteria merely give a structure to a comprehensive report and make different assessments comparable. Furthermore, one should not too hastily regard the lack of a feature as a definite deficiency, since it may be compensated to a degree by some design decision elsewhere in the system. A system has to be assessed as a whole, not by its individual details.

Finally, the domain framework and the assessment criteria help increasing general awareness of the various aspects of customisable method support environments. Without doubt, there are currently no systems in the market nor as academic prototypes that are comprehensive in terms of these criteria. The results of this study thus have remarkable potential contribution to further development of such systems.

References

Bandinelli, S., Fuggetta, A. & Ghezzi, C. 1993. Software Process Model Evolution in the SPADE Environment. IEEE Transactions on Software Engineering, 19, 12, 1128-1144.

Booch, G, Rumbaugh, J. & Jacobson, I. 1999 The Unified Modeling Language: User Guide. Reading, MA: Addison-Wesley.

Brinkkemper, S., Saeki, M. & Harmsen, F. 1999. Meta-Modelling Based Assembly Techniques for Situational Method Engineering. Information Systems, 24, 3, 209-228.

Conradi, R. & Jaccheri, M.L. 1993. Customization and Evolution of Process Models in EPOS. In: N. Prakash, C. Rolland and B. Pernici (Eds.) Information System Development Process. Amsterdam: Elsevier Science Publishers. 23-39.

Conradi, R. & Liu, Ch. 1995. Process Modelling Languages: One or Many? In W. Schäfer (Ed.) Software Process Technology, EWSPT'95, LNCS 913. Berlin: Springer-Verlag, 98-118.

Derniame, J.-C. & Kaba, A.B. 1999. Software Process: Principles, Methodology, and Technology. LNCS 1500. Berlin: Springer-Verlag.

Dowson, M. & Fernström, C. 1994. Towards Requirements for Enactment Mechanisms. In: B. Warboys (Ed.) Software Process Technology, EWSPT'94. LNCS 772. Berlin: Springer-Verlag.

Finkelstein, A., Kramer, J. & Nuseibeh, B. 1994. Software Process Modelling and Technology. New York: Wiley.

Froelich, G., Tremblay, J. & Sorenson, P. 1995. Providing Support for Process Model Enaction in the MetaView Metasystem. Proceedings of the 7th International Workshop on Computer-Aided Software Engineering, Toronto, Canada, July 10-14, 241 - 249.

Harmsen, F., Brinkkemper, S. & Oei, H. 1994. A Language and Tool for the Engineering of Situational Methods for Information Systems Development. In J. Zupancic, S. Wrycza (Eds.) Proceedings of the ISD'94 Conference, Bled, Slovenia, 206-214.

Jaccheri, M.L., Baldi, M. & Divitini, M. 1999. Evaluating the requirements for software process modeling languages and systems. Proceedings of the International Workshop on Process support for Distributed Team-based Software Development (PDTSD'99), Orlando, USA, July - August 1999.

Jarke, M. & Rose, T. 1992. Specification Management with CAD°. In P. Loucopoulos & R. Zicari (Eds.) Conceptual Modeling, Databases, and CASE. New York: Wiley, 489-505.

Jarke, M., Pohl, K., Rolland, C. & Schmitt, J.-R. 1994. Experience-Based Method Evaluation and Improvement: A process modeling approach. In: T.W. Olle, A.A. Verrijn-Stuart (Eds.) Proceedings of the IFIP WG8.1 Working Conference CRIS´94, Amsterdam: North-Holland, 1-27.

Kaipala, J. 2000. Integrating MetaCASE Environments by Using Hypertext. University of Jyväskylä. Computer Science and Information Systems Reports, Technical Reports TR-25. Licentiate thesis.

Kaiser, G.E. & Ben-Shaul, I.Z. 1993. Process Evolution in the Marvel Environment. In: W. Schaefer (Ed.), Proceedings of the 8th International Software Process Workshop. IEEE Computer Society Press, 104-106.

Kelly, S., 1998. Towards a Comprehensive MetaCASE and CAME Environment: Conceptual, Architectural, Functional and Usability Advances in MetaEdit+. University of Jyväskylä. Jyväskylä Studies in Computer Science, Economics and Statistics 41. PhD. Thesis.

Kelly, S., Lyytinen, K. & Rossi, M. 1996. METAEDIT+ — A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In P. Constantopoulos, J. Mylopoulos & Y. Vassiliou (Eds.) Advanced Information Systems Engineering, LNCS 1080. Berlin: Springer-Verlag, 1-21.

Koskinen, M. 1996a. Bringing Process Concepts Alive: on designing process modelling languages in a metaCASE environment. University of Jyväskylä. Master's thesis in Computer Science and Information Systems.

Koskinen, M. 1996b. Designing Multiple Process Modelling Languages for Flexible, Enactable Process Models in a MetaCASE Environment. In A.-H. Seltheit & B.A. Farshchian (Eds.) Proceedings of the 7th Workshop on the Next Generation of CASE Tools, Heraklion, Crete, Greece, May 1996. Trondheim, Norway: Norwegian University of Science and Technology, (no page numbers).

Koskinen, M. 1999. A Metamodelling Approach to Process Concept Customisation and Enactability in MetaCASE. University of Jyväskylä. Computer Science and Information Systems Reports, Technical Reports TR-20. Licentiate thesis.

Koskinen, M. 2000a. Conceptual Foundations of Process Metamodelling. Submitted for publication.

Koskinen, M. 2000b. Toward Customisation of Process Modelling Languages in Computer Aided Process Engineering. Submitted for publication.

Koskinen, M. & Marttiin, P. 1997. Process Support in MetaCASE: Implementing the Conceptual Basis for Enactable Process Models in MetaEdit+. In: J. Ebert, C. Lewerentz (Eds.) Software Engineering Environments. IEEE Computer Society Press, 110-123.

Koskinen, M. & Marttiin, P. 1998. Developing a Customisable Process Modelling Environment: Lessons Learnt and Future Prospects. In: V. Gruhn (Ed.) Proceedings on the 6th European Workshop on Software Process Technology, EWSPT'98. LNCS 1487. Springer-Verlag. 13-27.

Koskinen, M. & Marttiin, P. 2000. Comparing Two Traditions: Towards an Integrated View of Method Engineering and Process Engineering. To be submitted to European Journal of Information Systems.

Kumar, V.S. 1995. Personal Software Process in Meta-CASE. CMPT 856 - Project. Url at http://www.cs.usask.ca/grads/vsk719/academic/856/ project/project.html. Accessed on 18.5.2000.

Laamanen, P. 1995. Automation of Software Product Metrics: A Proposal for a Metamodel Based Metrics Engine. In: I. Mitchell, I. Ferguson & N. Parrington (eds.) The First International Conference on MetaCASE, 5-6 January, Sunderland, UK. Sunderland, UK: The University of Sunderland. (no page numbers).

Lonchamp, J. 1995. CPCE: A Kernel for Building Flexible Collaborative Process-Centered Environments. In: Software Engineering Environments. IEEE Computer Society Press, 28-41.

Luoma J. & Somppi, M. 1996. Concurrency Control in Multi-User MetaEdit+, (in Finnish Samanaikaisuuden hallinta monen käyttäjän MetaEdit+:ssa). University of Jyväskylä. Master's thesis in Computer Science and Information Systems.

Lyytinen, K., Marttiin, P., Tolvanen, J.-P., Jarke, M., Pohl, K. & Weidenhaupt, K. 1998. Bridging the Islands of Automation. In: S.T. March & J. Bubenko Jr. (eds) Proceedings of the Eight Annual Workshop on Information Technologies and Systems (WITS'98). University of Jyväskylä. Computer Science and Information System Reports, Technical Reports TR-19.

Marttiin, P. 1994. Towards Flexible Process Support with a CASE Shell. In G. Wijers, S. Brinkkemper, T. Wasserman (Eds.) Advanced Information Systems Engineering, LNCS 811. Berlin: Springer-Verlag, 14-27.

Marttiin, P., Harmsen, F. & Rossi, M. 1996. A Functional Framework for Evaluating Method Engineering Environments: the case of Maestro II/ Decamerone and MetaEdit+. In S. Brinkkemper, K. Lyytinen & R.J. Welke (Eds.) Method Engineering: Principles of method construction and tool support, London: Chapman & Hall, 63-86.

Marttiin, P. 1997. Can process-centred environments provide customised process support needed in metaCASE? A literature review. In G. Grosz (Ed.) Proceedings of the 1st International Workshop on the Many Facets of Process Engineering. Gammarth, Tunis, 165-180.

Marttiin, P. 1998a. Customisable Process Modelling Support and Tools for Design Environment. University of Jyväskylä. Jyväskylä Studies in Computer Science, Economics and Statistics 43. PhD Thesis.

Marttiin, P. 1998b. How to Support CASE Activities through Customisable Process Models: Experiments of CPME/MetaEdit+ Using VPL Formalism and ISPW-6 Example. In M. Khosrowpour (Ed.) Computer Supported Organizational Work: Proceedings of the 5th International Conference on Software Process. Hershey: Information Resources Management Association.

Marttiin, P., Lyytinen, K., Rossi M., Tahvanainen V.-P., Smolander K. & Tolvanen, J.-P. 1995. Modeling Requirements for Future CASE: modeling issues and architectural considerations. Information Resource Management Journal, 8, 1, 15-25.

McLeod, G. 1995. A Meta-CASE Tool Implementing a Generic Method Model for Representation, Integration and Management of Methods. In: I. Mitchell, I. Ferguson & N. Parrington (eds.) The First International Conference on MetaCASE, 5-6 January, Sunderland, UK. Sunderland, UK: The University of Sunderland. (no page numbers).

Mi, P. & Scacchi, W. 1992. Process Integration in CASE Environments. IEEE Software, March, 45-53.

Mi, P. & Scacchi, W. 1996. A Meta-Model for Formulating Knowledge-Based Models of Software Development. Decision Support Systems, 17, 3, 313-330.

Nissen, H., Jeusfeld, M., Jarke, M., Zemanek, G. & Huber, H. 1996. Managing multiple requirements perspectives with metamodels. IEEE Software, March, 37-48.

Oinas-Kukkonen, 1997. Improving the functionality of software design environment by using hypertext, Department of Information Processing Science, Oulu University Press, Ph.D. Thesis.

Phalp, K. & Shepperd, M. 1994. A Pragmatic Approach to Process Modelling, In: B. Warboys (Ed.) Software Process Technology, EWSPT'94. LNCS 772. Springer-Verlag. 65-68.

Pohl, K. 1996. Process-Centered Requirements Engineering. New York: Wiley.

Pohl, K., Weidenhaupt, K. Dömges, R., Haumer, P., Jarke, M., & Klamma, R. 2000. PRIME – Toward Process Integrated Modeling Environments. ACM Transactions on Software Engineering and Methodology, 8, 4, 343-410.

Rolland, C., Plihon, V. & Ralyté, J. 1998. Specifying the Reuse Context of Scenario Method Chunks. In: B. Pernici & C. Thanos (eds.) Advanced Information Systems Engineering, CAiSE'98. LNCS 1413. Berlin: Springer-Verlag, 191-218.

Rolland, C. & Prakash, N. 1993. Reusable Process Chunks. In: W. Marík, J. Lazanský & R. Wagner (eds.) Database and Expert Systems Applications, DEXA'93. LNCS 720. Berlin: Springer-Verlag, pp. 655-666.

Rolland, C., Souveyet, C. & Moreno, M. 1995. An approach of defining ways-of-working. Information Systems, 20, 4, 337-359.

Rossi, M. 1998. Advanced Computer Support for Method Engineering: Implementation of CAME Environment in MetaEdit+. University of Jyväskylä. Jyväskylä Studies in Jyväskylä Studies in Computer Science, Economics and Statistics 42. Ph.D. Thesis.

Rossi, S. & Sillander, T. 1998a. A Software Process Modelling Quest for Fundamental Principles. In R. Walter & J. Baets (Eds.) Proceedings of the 6th European Conference on Information Systems (ECIS). Euro-Arab Management School, Spain, 557-570.

Rossi, S. & Sillander, T. 1998b. A Practical Approach to Software Process Modelling Language Engineering. In V. Gruhn (Ed.) Proceedings on the 6th European Workshop on Software Process Technology, EWSPT'98, LNCS 1487. Springer-Verlag, 28-42.

Scacchi, W. 1996. Modeling, Simulating, and Enacting Complex Organizational Processes: A Life Cycle Approach. In K. Carley, L. Gasser & M. Prietula (Eds.) Simulating Organizations: Computational Models of Institutions and Groups. AAAI Press/MIT Press, 153-168.

Sharp, H., Woodman, M., Hovenden, F. & Robinson, H. 1999. The Role of 'Culture' in Successful Software Process Improvement. In: G. Chroust (ed) Proceedings of the 25th Euromicro Conference (EUROMICRO '99), Milan, Italy, September 8-10.

Si-Said, S., Rolland, C. & Grosz, G. 1996. MENTOR: A Computer Aided Requirements Engineering Environment. In P. Constantopoulos, J. Mylopoulos & Y. Vassiliou (Eds.) Advanced Information Systems Engineering, LNCS 1080. Berlin: Springer-Verlag, 22-43.

Skelton, J. 1995. MetaCASE and Software Process Maturity. In: I. Mitchell, I. Ferguson & N. Parrington (eds.) The First International Conference on MetaCASE, 5-6 January, Sunderland, UK. Sunderland, UK: The University of Sunderland. (no page numbers).

Smolander, K. 1992. OPRR - A Model for Methodology Modeling. In K. Lyytinen & V.-P. Tahvanainen (Eds.) Next Generation of CASE Tools, Studies in Computer and Communication Systems. Amsterdam: IOS press, 224-239.

Smolander, K., Tahvanainen, V.-P. & Lyytinen, K. 1990. How to Combine Tools and Methods in Practice — a Field Study. In B. Steinholtz, A. Sølvberg & L. Bergman (Eds.) Advanced Information Systems Engineering LNCS 436. Berlin: Springer-Verlag, 195-211.

Sorenson, P.G., Tremblay, J-P. & McAllister, A.J. 1988. The Metaview system for many specification environments. IEEE Software, 30, 3, 30-38.

Sutton, S.M., Tarr, P.L. & Osterweil, L.J. 1995. An Analysis of Process Languages. University of Massachusetts, Department of Computer Science. CMPSCI Technical Report 95-78.

Wijers, G. 1991. Modeling Support in Information Systems Development. Amsterdam: Thesis publishers, Ph.D. Thesis.

Zhang, Z. 2000. Defining Components in a MetaCASE Environment. In: B. Wangler & L. Bergman (eds.) Advanced Information Systems Engineering, CAiSE 2000. Berlin: Springer-Verlag, pp. 341-354.

YHTEENVETO (FINNISH SUMMARY)

Menetelmäkehityksen tavoitteena on tuottaa tietojärjestelmiä ja ohjelmistoja tuottaville organisaatioille yksilöllisesti sovitettuja menetelmiä. Tällaisessa menetelmässä voidaan tarkastella niin haluttujen tuotosten sisällöllisiä ja rakenteellisia ominaisuuksia, tietojärjestelmien ja ohjelmistojen kehittämisprosessia, kuin tähän prosessiin osallistuvien toimijoiden rooleja ja organisointiakin.

Tämän työn erityisala liittyy prosessien mallintamiseen. Tarkastelun kohteena on prosessimallinnuskielten määrittely, sovittaminen ja toteutus tietokonepohjaisessa systeemityön tukiympäristössä nk. prosessin metamallinnuksen keinoin. Työssä kehitetään tähän liittyvää teoriaa ja käsitteellistä perustaa sekä selvitetään teorian soveltamiseen liittyviä seikkoja tietokoneavusteisessa menetelmäkehityksessä.

Työssä käsitellyt aiheet jakautuvat neljään ryhmään: 1) prosessimallinnuskielten muokkaamiseen ja näillä kielillä tuotettujen prosessimallien suorittamiseen soveltuva ohjelmistoarkkitehtuuri; 2) prosessimallinnuskielten muokkaamisen tavat ja periaatteet sekä prosessimallien suorittamista tukevien mekanismien periaatteet; 3) prosessimallinnuskielten muokattavuuden edellyttämät käsitteelliset rakenteet ja niiden toteuttaminen tietokonetukeen; 4) prosessimallien suorittamista tukevat mekanismit ja niiden toteuttaminen tietokonetukeen.

Suuri osa työn teoriaosaan liittyvästä kontribuutiosta koostuu erilaisista luokitteluista sekä tutkittavien aiheiden ja niiden taustan filosofisesta ja käsitteellisestä selvittämisestä. Nämä tarkastelut kohdistuvat ohjelmistoarkkitehtuuriin, mallinnuskielten ja -tekniikoiden luonteeseen sekä metamallintamiskäsityksiin. Keskeisenä kiinnostuksen aiheena on kuitenkin kehittää prosessin metamallinnuksen teoriaa ja sen kautta tietokoneavusteinen keino muokata prosessimallinnuskieliä. Työn konstruktiiviseen osaan sisältyy prototyyppisen, prosessin metamallintamista ja prosessien suoritusta tukevan ohjelmiston suunnittelu ja toteutus. Lisäksi työn kontribuutiona on joukko kriteerejä joiden avulla muokattavia menetelmätukiympäristöjä voidaan tarkastella ja arvioida.

Tutkimusmetodologiana on konstruktiivinen lähestymistapa, jossa tutkimus etenee vähittäin ja iteratiivisesti. Sen iteratiivisina vaiheina ovat havainnointi, teoriakehitys, järjestelmäkehitys, ja kokeilut. Tutkimuksen pääpaino on teoriakehityksessä, jota muut vaiheet tukevat.